Tampereen yliopisto

Daniel Kullberg

MALLIPOHJAISEN TESTAUKSEN HYÖDYNTÄMINEN MOBIILISOVELLUK-SESSA

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Toukokuu 2019

TIIVISTELMÄ

Daniel Kullberg: Mallipohjaisen testauksen hyödyntäminen mobiilisovelluksessa
Diplomityö
Tampereen yliopisto
Tietotekniikka
Toukokuu 2019

Testaus on tärkeä osa ohjelmistojen kehitystä. Ilman kattavaa testaus ei voida sanoa toimiiko kehitetty ohjelmisto oikein. Testauksen perimmäinen tarkoitus on löytää mahdollisimman paljon virheitä, jotta ne pystytään korjaamaan kehitysvaiheessa. Testausta voi tehdä perinteisesti manuaalisesti tai automaattisesti testiskriptejä hyödyntäen. Testiautomaatio on viime aikoina kehittynyt paljon ja tuottanut monia uusia ideoita sekä kehittänyt vanhoja ideoita pidemmälle. Yksi tällainen idea on mallipohjainen testaus.

Työssä lähdettiin tutkimaan miten mallipohjaista testausta voisi hyödyntää ML Rahat -mobiilisovelluksen testauksessa. Mallipohjainen testaus käyttää sovelluksesta tehtäviä malleja, joiden mukaan testitapaukset luodaan automaattisesti algoritmeilla. Näitä testitapauksia voidaan ajaa automaattisesti, kun mallin pohjilta on toteutettu testit Robot Frameworkillä.

Mallipohjainen testaus toisi automaatiotestauksen lähemmäksi sovelluksen kehittämisvaihetta. Kun automaatiotestaus ja automaatiotestien tekeminen ei ole lähellä kehitystä niin niiden kyky testata sovellusta kärsii eikä niistä saada kaikkea hyötyä irti. Mallipohjainen testaus toisi tähän apua, kun sovelluksen malli ja sen pohjalta toteutettavat testit tehtäisiin testaajan toimesta samanaikaisesti sovelluksen kehittämisen kanssa, samassa vaiheessa, kun manuaalitestausta tehdään. Mallipohjainen testaus toisi myös apua sovelluksen dokumentaatioon ja määritelmiin mallien muodossa. Testaajien tekemät testausmallit toimivat samalla myös dokumentaationa siitä, miten sovelluksen tulisi toimia ja mitä toimintoja sovelluksessa on missäkin vaiheessa mahdollisia.

Työssä toteutettiin mallipohjainen testaus mobiilisovellukselle. Sovelluksen pohjalta tehtiin ensin mallit, joiden pohjilta luotiin automaattisesti erinäisiä testitapauksia. Mallin pohjilta toteutettiin automaatiotestit Robot Frameworkillä, joilla luotuja testitapauksia ajettiin. Lopputuloksena on kokonaisvaltainen ja kattava testimenetelmä, jolla voi testata sekä kehityksen aikaista sovellusta, että käyttää myös regressiotestauksessa.

Mallipohjainen testaus vaikuttaa lupaavalta menetelmältä. Sen avulla on mahdollista tuottaa kattavat testit ja dokumentaatiota itse sovelluksen toiminnasta. Mallipohjainen testaus on hieman työläämpää kuin manuaalinen testaus tai mitä tämän hetkinen automaatiotestausprosessi on. Kuitenkin sen hyödyt ovat suuremmat kuin työmäärän kasvu, joten sen käyttöä ja kokeilua tulisi jatkaa ottamalla se mukaan kehityksen aikaiseen testaukseen.

Avainsanat: mallipohjainen testaus, mobiilisovellus, Android

ABSTRACT

Daniel Kullberg: Model-Based Testing in a Mobile Application
Master of Science Thesis
Tampere University
Information Technology
May 2019

Testing is an important part of software developing. Without comprehensive testing, it is not possible to state whether the developed software is working correctly. The basic principle of testing is to find as many defects as possible so they can be fixed in the developing phase. Testing can be traditionally done manually or automatically using testing scripts. Test automation has lately evolved a lot and has produced many new interesting ideas and developed old ideas further. One of these ideas is model-based testing.

In this thesis, it is explored hoe model-based testing can be utilized in the testing of ML Money mobile application. Model-based testing uses models that are created from the application. Test cases are automatically created from these models. These test cases can be run automatically when tests from the models have been implemented in Robot Framework.

Model-based testing would bring test automation closer to the developing stage of the application. When test automation and creating its test are not close to developing, their ability to test the application suffers and test automation cannot fulfill its potential. Model-based testing would fix this as the model of the application and its tests based on it would be done by the tester at the same time when the application is being developed. This is the same stage manual testing is done. Model-based testing would also help in the documentation and specifications of the application with models. Testing models created by testers would work as documentation how the application should work and what functions are available at some state.

Model-based testing was implemented to mobile application in this thesis. First models were created based on the application. These models were used to create test cases. Robot Framework was used to implement the automated tests that were used to run the test cases. End result is a comprehensive and encompassing testing method that can be used to test application during developing phase or in regression testing.

Model-based testing looks like a promising method. It can be used to create comprehensive tests and it also creates documentation of the functionality of the application. Model-based testing requires more work than manual testing or the current test automation method. However, its advantages outweigh the increase in work so it should be used and tried out more by taking it into use during the developing phase of the application.

Keywords: model-based testing, mobile application, Android

ALKUSANAT

Tämä diplomityö toteutettiin Mandatum Lifelle tutkimuksena. Ideana on tutkia miten yhtiö mobiilisovelluksen testausta voisi parantaa, ja ratkaisuehdotukseksi valikoitui mallipohjainen testaus.

Haluan kiittää kaikki Mandatum Lifeä tuesta työssä, kaikkia Mandatum Lifellä siitä, että tekevät työpaikasta ja työstä mukavaa tehdä ja mielekkään. Haluan erityisesti kiittää esimiestäni Antti Lehikoista, joka on koko diplomityöprosessin aikana osoittanut mielenkiintoa työtäni kohtaan sekä tukenut sen tekemistä. Haluan myös kiittää Tampereen Yliopistoa, erityisesti professori Hannu-Matti Järvistä tuesta ja ohjauksesta työn tekemisen aikana.

Haluan kiittää vanhempiani, jotka ovat kasvattaneet minut ja tukeneet minua opiskeluissani. Suurin kiitos kuuluu tulevalle vaimolleni Emilia Roivakselle, joka on tukenut minua diplomityöprosessin joka vaiheessa ja auttanut minua löytämään motivaatiota silloin kun se on itselläni ollut vähissä.

Espoossa, 22.5.2019

Daniel Kullberg

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

SUT	engl. System Under Test, testauksen kohteena oleva ohjelmisto
FSM	engl. Finite State Machine, tilakone
IAB	engl. In-App Browser, sovelluksen sisään avattava selain
AVD	engl. Android Virtual Device, virtuaalinen Android-laite
SAFe	engl. Scaled Agile Framework, ketterä kehitysmenetelmä

1. JOHDANTO

Testaus on tärkeä osa sovelluksen kehityskaarta. Kuitenkin kokemuksen mukaan usein juuri testauksen kohdalta aikataulu antaa periksi, ja testausaika supistuu. Ehkä tähän on osin syynä se, että testauksen päätavoite on löytää sovelluksesta virheitä, mikä näkyy usein lisätyönä. Eli onnistuessaan testaus tuottaa lisää työtä. Tämä työ kerääntyy usein sovelluksen kehityksen loppuvaiheeseen, kun sovellus on lähes valmis. Voisiko testaus siis tehostaa jotenkin, että se tuottaisi tasaisemmin työtä jo aikaisessa vaiheessa sovelluksen kehitystä.

Perinteisesti testaus on ollut manuaalista työtä missä testaaja käy sovellusta läpi ja etsii virheitä. Apuna on myös automaatiotestaus, yksikkötestien tai testiskriptien muodossa. Voisiko jotenkin yhdistää molemmista parhaat puolet, manuaalitesteistä ihmisen silmä ja suunnitelmat, sekä automaatiotestien helppo toistettavuus ja uudelleen ajo? Yksi ratkaisu tähän voisi olla mallipohjainen testaus.

Mallipohjainen testaus hyödyntää algoritmeja testauksessa perinteisen testausmenetelmien kanssa luoden nykyaikaisen ja modernin testausmenetelmän. Algoritmien hyödyntäminen takaa sen, että menetelmä on kattava ja skaalautuva mikä on tarpeen varsinkin monimutkaisissa ohjelmistoissa. Mallipohjainen testaus ei ole menetelmänä uusi. Se on kuitenkin vasta hiljalleen kerännyt suosiota viime vuosina testaajien parissa ja on yleisesti kuvattu olevan seuraava iso juttu automaatiotestauksessa. Vielä mallipohjainen testaus ei ole kuitenkaan lyönyt itseään läpi.

Tässä työssä tutkitaan, miten mallipohjainen testaus soveltuu mobiiliapplikaatioiden testaukseen toteuttamalla menetelmä esimerkkisovellukseen. Työ on siis konstruktiivinen tutkimus, joka perustuu tavoitteisiin. Työn tavoitteena on saada riittävästi tietoa siitä, onko mallipohjainen testausmenetelmä kannattavaa ja kannattaako sen käyttöä laajentaa sekä työn kohteena olevassa sovelluksessa sekä muihin sovelluksiin.

Työ alkaa luvun 1 johdannolla, jonka jälkeen luvussa 2 käydään mallipohjaisen testauksen teoriaa läpi. Luvussa 3 esitellään sovellus, johon toteutetaan mallipohjainen testaus. Luvussa 4 toteutetaan mallit sovellukseen ja luvussa 5 malli kokeillaan ja esitellään mitä tuloksia mallin pohjilta ajetuista testitapauksista saatiin. Luvussa 6 käydään läpi tulokset ja analysoidaan niitä. Mallin pohjilta saatavia testitapauksia vertaillaan myös kattavuuden osilta nykyisiin automaatiotesteihin tässä luvussa. Työn päättyy lukuun 7, jossa on yhteenveto.

2. MALLIPOHJAINEN TESTAUS

Ohjelmistojen testaaminen on pohjimmiltaan sitä, että ohjelmaan tai ohjelmistoon tuodaan jokin ärsyke, joka aiheuttaa jonkin toiminnon ohjelmistossa. Tämän toiminnon tulosta tarkastellaan ja verrataan johonkin ennalta oletettuun arvoon. Manuaalisessa testauksessa ärsykkeen aiheuttaa testaaja, automaatiotestauksessa automaatiokehys. [1] Mallipohjaisessa testauksessa ärsykkeet luodaan automaattisesti sovelluksen pohjalta luodun mallin avulla.

2.1 Teoria

Mallipohjainen testaus on ohjelmiston testausta, jossa testitapaukset luodaan osittain tai kokonaan mallin avulla. Malli kuvaa osaa tai kaikkia ominaisuuksia testattavasta ohjelmistosta. Testauksen kohteena oleva ohjelmisto, System Under Test (SUT) voi olla yksinkertaisuudessaan jokin yksittäinen metodi tai luokka koodissa, tai monimutkaisuudessaan kokonainen ohjelmisto, joka koostuu useasta järjestelmästä. Testauksen avuksi malli tuottaa käyttäytymiskuvauksen testattavasta ohjelmistosta. Tätä kuvausta käyttäen voidaan tuottaa testisettejä, joista voidaan päätellä, toimiiko testattava ohjelmisto haluttujen ominaisuuksien mukaisesti. [2]

Mallipohjainen testaus on siis testausta, jossa valmiiden testitapausten sijaan käytetään varsinaisesta sovelluksesta luotua mallia, josta saadaan luotua automaattisesti testitapauksia. Malli itsessään on yksinkertaistettu versio sovelluksesta, joka kuvaa vaikkapa mitä siirtymiä ja tiloja on mahdollista suorittaa mallinnettavassa ohjelmistossa tai kuvaa ohjelmiston, joitain prosesseja ja miten ne suoritetaan. [3]

Yksinkertaisimmillaan malli on graafinen esitys testauksen kohteena olevasta ohjelmistosta, esimerkiksi UML-kaavio, johon on kuvattu siirtymät ja syötteet mitä ohjelmistoon on mahdollista syöttää. Kun järjestelmän käyttäytyminen on ymmärretty ja kuvattu malliin, voidaan mallia käyttää testien luomiseen, jotka varmistavat tämän käyttäytymisen. [4]

Mallin voi luoda joko kehityksen kanssa samaan aikaan tai jälkikäteen jo valmiiseen sovellukseen. Jos malli luodaan jo mahdollisimman aikaisessa vaiheessa, on siitä tällöin apua myös kehittämisessä, esimerkiksi dokumentaation ja ohjelmiston toiminnan hahmottamisen muodossa. Jos malli tehdään jälkikäteen, on se enemmän testauksen apuna, kertomalla mahdollisia testitapauksia ja siirtämällä tietoa testaajalta toiselle. [5]

Mitä aikaisemmin malli tehdään, sitä aikaisemmin voidaan alkaa huomaamaan virheitä testauksen kohteena olevasta sovelluksesta. Kaikki virheet eivät johdu koodauksesta ja sovelluksen toteutuksesta, vaan myös määrittelyssä voi ilmetä virheitä. Siksi malli olisi hyvä saada tehtyä jo alkuvaiheessa, jotta mahdolliset virheet saadaan löydettyä dokumentaatiosta ja määrittelystä. Kun malli tehdään aikaisin, voidaan se tehdä puhtaasti määrittelyiden pohjilta. Kun malli tehdään myöhemmin, saatetaan sitä tehdä jo sovelluksen toiminnan pohjalta. Tällöin sovelluksen virheellinen toiminta saattaa kuvautua malliin asti. [3]

Nykytilanteessa kehittäjä luo ominaisuuden, tekee yksikkötestit, asentaa ohjelmiston tai sen uuden toiminnollisuuden testiympäristöön. Yleensä automaattitestit ajetaan tässä välissä. Testaaja testaa ja hyväksyy ohjelmiston toiminnan, jonka jälkeen myöhemmin ohjelmisto asennetaan tuotantoon. Mallipohjainen testaus sijoittuisi testaajan tekemän testauksen kohdalle ja toimisi rinnakkain manuaalitestauksen kanssa. Automaattitestit, joita nyt käytetään regressiotestauksessa ovat staattisia luonteeltaan, ne eivät mukaudu, ellei niitä päivitetä. Mallipohjainen testaus puolestaan on mukautuvaa, testit voivat pysyä samoina tai muuttua. Nykyiset automaattitestit ovat regressiotestejä. Jos ne löytävät jonkin virheen, se korjataan, jolloin testit muuttuvat varmistaviksi. Ne eivät auta siis uuden ominaisuuden testauksessa, vaan ovat vain varmistamassa, että vanha toiminnallisuus toimii edelleen. Mallipohjaista testausta voisi käyttää uuden kehityksen rinnalla testaamassa uutta toiminnallisuutta. [5]

2.1.1 Malli

Malli on yksinkertaistettu versio sovelluksesta, ikään kuin rautalankaversio tai runko sovelluksesta, sen ääriviivat. Malli sisältää osittain saman logiikan kuin varsinainen sovellus, mutta pelkistetysti. Malleja voi olla useita eri kehityksen kohteille, mutta yksi tärkeimmistä on testauksen käyttämä malli. Siihen on kuvattu testattavan sovelluksen toiminta sillä tasolla, kun se testausta varten on tarpeen. Joitain toimintoja on voitu pelkistää yhteen, kun taas toisia on voitu avata useammaksi vaiheeksi riippuen siitä miten kyseinen ominaisuus testataan. [5]

Ihmiselle on luontevaa piirtää monimutkaisesta tilanteesta kuva selkeyttämistä varten. Samaa tapaa voisi käyttää testauksen apuna ymmärtämään testauksen kohteena olevan sovelluksen toimintaa. Toiminnan visualisointi auttaa hahmottamaan ohjelmiston toimintaa, varsinkin jos se on monimutkainen. [3]

Malli auttaisi siis myös manuaalitestauksessa. Vaikka mallin tarkoitus on olla pohjana automaattisille testitapauksille voi samoja testejä testata myös manuaalisesti. Malli voi

olla myös yksi totuus sovelluksen oletetulle toiminnalle. Se on yksiselitteinen ja ratkaisee tilanteet, jossa eri ihmiset muodostavat oman käsityksensä siitä, miten ohjelmisto toimii tai miten sen tulisi toimia. Kun toiminnallisuus on kuvattu malliin, on kaikilla yhteinen käsitys ohjelmiston toiminnasta. [1]

Yksi vääjäämätön rajoitus mallipohjaisessa testauksessa on, että sen avulla johdetut testitapaukset ovat vain niin hyviä kuin mikä tieto on mallissa ollut. Tästä syystä on mallin tekoon kiinnitettävä paljon huomiota ja tarkkuuta. Koska malli on ihmisen tekemä, voi myös siinä olla virheitä. Siksi on tärkeää rajata mitä pidetään virheenä itse sovelluksessa ja mikä taas on virhe mallissa. [1]

Mallin on myös oltava abstraktimpi kuin testauksen kohteena oleva sovellus. Jos malli sisältää liikaa logiikkaa itse sovelluksesta, joudutaan testaamaan myös itse mallia [5]. Malliin tulee siis kuvata vain perusominaisuuksia, sekä pelkistää sitä kuvaamaan vain perustapaukset testattavasta sovelluksesta. [6]

Malleja on myös eri tyyppisiä. On muun muassa prosessimalleja, jotka kuvaavat sovelluksen toimintaa korkealla tasolla määrittelyvaiheessa ja sitten on testausmalli, joka kuvaa sovelluksen toimintaa testausmielessä. Onkin siis päätettävä, millainen malli halutaan. Voidaan myös tehdä monia malleja, jos koko prosessi on mallipohjainen. Tällöin ensimmäisiä malleja voidaan käyttää jälkimmäisten mallien teossa apuna. Kuitenkin on todettu, että testaajat saavat tehtyä paremman mallin testaustarkoitukseen myös ilman apuna, ja usein se malli on jopa parempi, kun se tehdään puhtaasti dokumentoinnin perusteella, eikä valmiin mallin pohjalta, joka voi jo sisältää jotain oletuksia tai virheitä itsessään. Malli tulee suunnitella siten, että se voi täyttää halutut testikriteerit, esimerkiksi käydään kaikki siirtymät läpi tai kaikki tilat läpi. Tässä työssä käytetään tilakonemallia (FSM), jonka kuvaa sovelluksen toimintaa tiloina ja siirtyminä. Tiloja voivat olla esimerkiksi sovelluksen sivuja, ja siirtymiä linkit ja napit, joita painamalla eri sivujen välillä liikutaan. [1]

2.1.2 Testauksen kohteena oleva sovellus

SUT on se sovellus, joka on testauksen kohteena. Malli toimii testaustyökalun ja SUTin välissä ja ikään kuin kertoo testaustyökalulle, mitä testitapauksia ajetaan. Tehty malli perustuu SUTiin ja on ikään kuin pelkistetty versio siitä. [1]

SUTin pitää olla tietynlainen, jotta tukee mallipohjaista testausta, pitää olla tarpeeksi monimutkainen, että malleista on hyötyä mutta ei liian, jotta ei tule liian monta mallia. Ei myöskään liian yksinkertainen, että mallipohjaisuudesta on jotain hyötyä. Mallin tyyppi

riippuu myös testauksen kohteena olevasta sovelluksesta. Useimmat mallit kuvaavat SUTin toimintoja, mutta ne voivat kuvata myös sen rakennetta tai sen dataa. [3]

2.1.3 Testien luominen mallista

Mallista luodaan testit jonkin mallipohjaisen testityökalun avulla. Se osaa luoda testitapaukset, jotka voidaan joko suoraan ajaa tai integroida automaatiotestaustyökaluun ajettavaksi, riippuen käytettävästä työkalusta. Tapoja on pääosin kaksi, online ja offline. Online menetelmässä testitapauksia luodaan samalla kun ohjelmistoa testataan. Seuraava askel testitapauksessa ei siis välttämättä ole tiedossa ennen kuin edellinen askel on suoritettu. Offline-metodissa taas koko testipolku luodaan ensin, jota ajetaan sitten myöhemmin askel kerrallaan. Koko testitapaus on siis etukäteen tiedossa, ennen kuin testiä lähdetään suorittamaan. [3][5]

Online-menetelmä voisi sopia paremmin kehityksen aikaiseen testaukseen ja on enemmän suunnattu kehittäjille. Online-metodissa ollaan riippuvaisia siitä, että työkalu tukee samaa ohjelmointikieltä, jolla testattava ohjelmisto on toteutettu. Testit tulee toteuttaa usein samalla kielellä. Online-metodi onkin siis lähempänä kehittäjän yksikkötestejä ja testausta. [5]

Offline-menetelmän hyötyjä ovat työkalujen riippumattomuus [5]. Mallipohjaisen testauksen työkalu vain luo testitapauksen ja sen testaamiseen tarvittavat automaatiotestit tulee toteuttaa itse halutulla automaatiotestauskehyksellä. Heikkous on se, että koska testi on luotu etukäteen, ei voida testitilanteessa arvioida, mikä olisi hyvä seuraava askel, koska se on jo etukäteen päätetty [5]. Tutkivaa testausta ei siis voida harjoittaa.

Testitapauksia voi luoda niin monta kuin haluaa. Usein työkalu tarjoaa eri kriteereitä, miten testitapauksia luodaan mallista. Näitä voivat olla esimerkiksi se, että käydään kaikki halutut sovelluksen tilat läpi, kaikki halutut siirtymät sovelluksesta läpi tai se että saavutetaan jokin tietty tila sovelluksessa. Testitapaukset luodaan haluttujen kriteerien mukaan jollain algoritmilla, vaikkapa satunnaisesti tai etsimällä lyhimmän polun. [4]

Mallin avulla saadaan siis ainakin teoriassa luotua parempia ja kattavampia testitapauksia, kuin mitä testaajat ja kehittäjät pystyvät luomaan. Algoritmi tietää mallin perusteella mitä on mahdollista tehdä sovelluksessa ja luo testitapaukset sen mukaan. Lisäksi, kun malli on kerran tehty sovelluksesta, voi sen perusteella luoda loputtomiin testitapauksia. [1]

2.2 Yleiset käyttökohteet

Mallipohjaista testausta voidaan käyttää yleisesti manuaali- tai automaatiotestauksessa luomalla sen avulla automaattisesti testitapauksia, tai sitä voidaan käyttää etsimään tiettyjä virheitä ja miten niitä saadaan toistettua. Mallille voidaan kertoa tiettyjä ehtoja ja tarkkailla sovelluksen käyttäytymistä, esimerkiksi tuleeko muistivuotoja tai kaatuuko sovellus johonkin tiettyyn kohtaan. [1]

Mallipohjainen testaus on erityisesti hyödyksi, kun testataan SUTin kattavuutta. Kun testattava sovellus on mallinnettu, osaa mallipohjaisen testauksen työkalu luoda mallin pohjilta tarvittavat testitapaukset, jotta testeistä tulee mahdollisimman kattavat. Näin ollen saadaan testattua esimerkiksi verkkosivuston kaikki sivut. Mallin pohjilta voidaan luoda automaattisesti myös esimerkiksi sellaiset testitapaukset, joissa käydään joka sivulla tai joissa testataan kaikki toiminnot, jotka johtavat eri sivuille. [4]

Mallipohjainen testaus on myös hyödyksi, jos halutaan tutkia jotain erityistä tilannetta. Mallin avulla voidaan toistaa tarkasti samat olosuhteet ja tutkia, mitä tapahtuu. Esimerkiksi jos tutkitaan vaikkapa sovelluksen kaatumista, voidaan samoja olosuhteita toistaa helposti, kun kuljettu polku on tiedossa. Tähän polkuun voidaan tehdä joka ajolla pieniä muutoksia, jotta saadaan tarkemmin haarukoitua häiriön alkuperä. Mallin pohjita voidaan luoda myös esimerkiksi pitkiä testitapauksia ja ajaa niitä. Ajoa voidaan tarkkailla esimerkiksi sovelluksen kaatumisten tai muistivuotojen varalta sekä voidaan tutkia, miten sovellus toimii suuren kuorman alaisena ajamalla monia testitapauksia stressitestien muodossa. [3][5]

Kun malli on luotu, voidaan sen pohjilta luotuja testejä ajaa loputtomiin, vaikka koko ajan. Tämä voi olla hyödyllistä, jos tutkitaan esimerkiksi muistivuotoja. Testit voidaan laittaa suoritukseen yön yli ja katsoa aamulla sovelluksen tilanne. Testaaja voi myös tutkia ajoja aktiivisesti. Mallipohjaisen testauksen avulla testaaja saa paremman käsityksen sovelluksen toiminnasta, varsinkin jos se sisältää monimutkaisia siirtymiä. Mallipohjainen testaus ei tarjoa apua yksinkertaisissa sovelluksissa, tai ainakaan nähty vaiva työn suhteen ei ole hyvä saavutettuun hyötyyn nähden. [3][5]

Mallipohjaisessa testauksessa täytyy kiinnittää huomioita tapauksiin, joissa tulee huomaamattomia virheitä. Jos sovellus kaatuu, sen huomaa helposti. Jos se ei kaadu, voi tulla virhe, mitä ei suoraan huomaa manuaalitestauksessa, jossa testaaja toimii tuomarina, mikä on häiriö ja mikä ei. Ohjelmallisesti testattaessa voidaan tarkkaan määritellä kriteerit, joiden mukaan sovellus joko toimii halutusti tai ei. Mallipohjaisessa testauksessa tilat ja siirtymät voi ja tulee määritellä niin tarkasti, että voidaan olla vakuuttuneita sovelluksen toimimisesta halutusti, kun testipolku on saatu suoritettua. [1]

2.3 Käyttökohteet ja -tarkoitus

Mallipohjaisella testauksella saadaan selkeämpiä testitapauksia, kun nämä eivät ole enää testaajan vastuulla. Malli ja mallipohjaisen testauksen työkalun avulla pystytään luomaan kattavat testit, joka sisältävät esimerkiksi kaikki siirtymät tai kaikki tilat [1]. Testaajan ei näin tarvitse käyttää paljon aikaa, jotta saisi mietittyä kattavat testitapaukset, jossa kaikki sivut tulisi käytyä läpi koska malli osaa tehdä sen automaattisesti ja tehokkaammin. Varsinkin suhteellisen monimutkaisissa sovelluksissa, joissa tiloja ja siirtymiä on monta, voi olla manuaalisesti hankalaa miettiä kattavia testitapauksia. [3][6]

Kun yksi malli on luotu, voidaan sen osasia soveltaa myös muihin sovelluksiin. Kun avainsanapohjaiset testit on yleensä nimetty tiettyyn tarkoitukseen, voidaan mallin yleiskäyttöisiä osia, kuten kirjautuminen, siirtää muualle. Mallia voi myös jakaa osiin pienempiin malleihin, jolloin näitä voidaan myös käyttää muissa sovelluksissa. Esimerkiksi jos toteutetaan kirjautumisen malli, voidaan samaa mallia käyttää myös muissa sovelluksissa pohjana, jos niissä ne sisältävät myös kirjautumisen. Malli voi olla siis osittain siirrettävä. [4]

Regressiotestauksessa mallipohjainen testaus voisi auttaa juuri offline-menetelmän avulla. Mallista voidaan luoda haluttuja testitapauksia, joita voidaan ajaa regressiona saman tyyliin kuten vaikka avainsanapohjaisia testejä. Jos sovellus muuttuu, on mallia muutettava ja siten myös mallin pohjilta luodut regressiotestit muuttuvat. Myös tavanomaisia regressiotestejä olisi tällöin muutettava, joten tässä mielessä mallipohjainen testaus ei eroa regressiona mitenkään nykytilanteesta. [3]

Mallipohjaista testausta voi käyttää myös integraatiotestauksen osana. Mitä aikaisemmin saadaan malli mukaan testaukseen, sitä aikaisemmin saadaan sovelluksen eri osa-alueita testattua, miten ne toimivat yhteen. Mallia voisi käyttää myös tutkimaan, kuinka valmis ohjelmisto on. Kun malli on luotu määritelmien pohjilta, tiedetään karkeasti, kuinka monta tilaa ja siirtymää tulee valmiissa sovelluksessa olemaan. Mallista luotujen testien perusteella voidaan sitten arvioida kuinka monta tilaa ja siirtymää on sillä hetkellä saavutettavissa. [3]

Mallipohjaista testausta voi siis hyödyntää missä vaiheessa tahansa testausta. Sitä voidaan käyttää yksikkötestaukseen online-menetelmällä, online- ja offline-menetelmiä systeemi ja hyväksymistestauksessa, sekä offline-menetelmää regressiotestauksessa. Kaikki testitasot vaativat vain mallin ja testit toteutetaan saman mallin pohjilta, jolloin koko ajan tulee testattua johdonmukaisesti sovellusta. [3]

2.4 Hyödyt ja haitat

Mallia voisi hyödyntää jo projektin suunnitteluvaiheessa, jolloin malli auttaa jo dokumentointi ja määrittelyvaiheessa ymmärtämään sovelluksen toimintaa. Usein projektin alkuvaiheessa suunnitellaan prototyyppi, joka edustaa näkemystä toiminnallisuudesta ja tyyleistä. Tämä prototyyppi kuitenkin kuvaa vain ennalta määrätyt toimenpiteet. Jos prototyypin korvaisi sovelluksen mallilla, saataisiin selkeämpi kuva heti projektin alkuvaiheessa millainen valmis sovellus olisi. Kun tiedettäisiin alustavalla tasolla esimerkiksi siirtymät ja tilat, saataisiin heti kehityksen aluksi selkeä runko tukemaan kehitystä. [3][4][5]

Kun malli on saatu tehtyä, on vaikein työ takana. Tämän jälkeen voidaan luoda haluttuja testitapauksia loputtomiin. Mallin yksi hyöty onkin, että se ei kulu ajossa. Sitä voidaan ajaa niin paljon kuin halutaan. Kun tavalliset automaatiotestit on luotu, on näitä testejä ajettava ja päivitettävä. Näillä testeillä saadaan testattua vain ne tapaukset, joihin testit on suunniteltu. Mallipohjaisella testauksella voidaan itse luoda tai antaa työkalun luoda automaattisesti kattavat testitapaukset. Näin saadaan testattua paljon kattavammin koko sovellus. Sen lisäksi että saadaan testattua halutut ominaisuudet, voidaan sovellusta testata kattavammin. [5]

Mallin työläisyys riippuu testauksen kohteesta. Mitä monimutkaisempi järjestelmä on testauksen kohteena, sitä monimutkaisempi myös mallista tulee [3]. Kun malliin on kuvattava kaikki sovelluksen toiminnollisuudet, tulee määritelmistä tärkeää, varsinkin jos mallia luodaan jälkikäteen jo valmiiseen projektiin. Mallin luominen voi olla myös sujuvaa, jos saatavilla on hyvät määritelmät, ja malli myös toimii osittain itse sovelluksen määrittelyinä. [5]

Mallin luominen voi olla työlästä. Mitä monimutkaisempi sovellus on testauksen kohteena, sitä monimutkaisempi myös mallista tulee. Mallin luominen ja sen testien toteuttaminen vaatii ohjelmointiosaamista, joten pelkälle testaajalle mallin luominen voi alussa tuottaa vaikeuksia. Tällöin mallin luominen tarvitsee aloittaa ohjelmoinnin opettelemisella. [5]

Malli tarvitsee lisäksi aputyökalun, joka osaa tuottaa testitapauksia automaattisesti. Malli on siis sidoksissa sekä testauksen kohteeseen että aputyökaluun. Tämän lisäksi tarvitaan vielä jokin testiautomaatiojärjestelmä ajamaan testitapaukset. Tämä voi olla myös etu, sillä malli saadaan integroitua olemassa oleviin järjestelmiin ja niiden tueksi, jolloin testien kattavuus paranee edelleen. [3]

Haittana on myös, että koska mallin luo ihminen, myös malli voi sisältää virheitä. Malli tulisikin luoda ideaalitilanteessa yhteistyössä testaajan ja määrittelyiden tekijöiden kanssa määritelmien pohjilta. Tällöin saadaan varmistettua, että malli vastaa mahdollisimman tarkasti määritelmiä, ja sitä voi käyttää testitapausten pohjana. Koska myös malli voi sisältää virheitä, tulee sen avulla löydetyt virheet tutkia, että onko kyse sovelluksen virheestä vai itse mallin virheestä. Joissain tilanteissa tämä ei välttämättä ole aina selkeää, ja virheiden päättely tuottaa lisätyötä. [3]

2.5 Mallipohjainen testaus mobiilissa

Tässä työssä oletuksena on, että mallipohjainen testaus mobiilisovelluksissa ei varsinaisesti eroa mitenkään työpöytäsovellusten testauksesta. Pääosin erot tulevat siitä, miten testejä ajetaan. Työpöytäsovellus tai verkkosivut on helppo testata, sillä niitä voi ajaa samalla tietokoneella, millä testejäkin ajetaan. Mobiilisovellukset pyörivät omissa laitteissaan, jotka pitää kytkeä tietokoneeseen tai jotenkin muuten saada yhteys niihin.

Mobiilisovelluksia ja laitteita on myös monia eri tyyppisiä. Jo alustoja on kaksi eriävää, Googlen Android ja Applen iOS. Mikäli halutaan testata molempia alustoja, on huolehdittava, että samat testit ja mallit toimivat molemmissa alustoissa. Alustojen lisäksi myös itse laitteita on monenlaisia, joiden ominaisuudet saattavat vaihdella paljonkin. Järkevää on siis rajata, ainakin aluksi, testaus tiettyyn alustaan ja laitteeseen. Sovellusten tulisi toimia eri laitteilla samalla tavalla. Mallipohjaisen testauksen yksi hyöty onkin, että mallien perusteella luodut testit tulisi toimia samalla tavalla niin kauan testi pysyy samana. Mobiilisovellusten testauksessa voidaankin siis eri laitteille ajaa mallipohjaisen testauksen avulla samoja testejä ja vertailla tuloksia. [7][8]

Mobiilisovellukset ovat koko näytön sovelluksia, tarkoittaen että koko laite ajaa vain yhtä sovellusta kerrallaan. Tämä voi olla etu verrattuna verkkosovelluksiin, sillä tietokoneella voi olla esimerkiksi monta eri sivua auki samaan aikaan. Kuitenkin usein myös niissä keskitytään yhteen sivuun kerrallaan testauksessa.

2.6 Soveltuvuus tässä työssä

Mandatum Lifessä on halua kehittää mobiilisovelluksen testausta ja erityisesti mobiilisovelluksen testausautomaatiota. Yhä enemmän ja enemmän asiakkaat ovat siirtymässä käyttämään tarjolla olevia palveluita mobiilissa, sekä alkavat myös vaatimaan uusia toiminnollisuuksia. Tästä syystä mobiilisovelluksen vakaa toiminta on tärkeää, eikä kaikkea

toiminnollisuutta pystytä kattavassa määrin testaamaan pelkästään manuaalisesti. Kattava automaatiotestaus on avain tähän ja mallipohjainen testaus voisi olla hyvä lisä mobiiliautomaatioon.

Mallipohjainen testaus voisi sopia hyvin Mandatum Lifen mobiilisovellukseen, sillä se on monimutkainen ja vaikea käydä käsin läpi koska dokumentaatio on osin puutteellista. Testitapausten ja -settien miettiminen ja keksiminen on siis vaikeaa. Kun ei ole tarkkaa tietoa sovelluksen toiminnasta, tulee testauksessa välillä epäselviä tilanteita vastaan. Mallipohjaisuus toisi apua sekä uuden että vanhan toiminnollisuuden testaukseen, kun sovelluksen toiminta kuvattaisiin malleihin. Malleista saataisiin tukea dokumentoinnin muodossa erityisesti testitapausten luontiin, sekä mallit auttaisivat visualisoimaan sovelluksen toimintaa, jolloin niistä olisi hyötyä myös manuaalitestauksessa.

Mallinpohjaisen testauksen avulla koko malliin kuvattu sovelluksen toimintaa voisi käydä läpi. Näin saataisiin erittäin kattavia testejä, jotka voisivat kestää pitkiäkin aikoja. Mallin voisi jättää ajamaan näitä testejä yksin pitkiäkin aikoja, esimerkiksi yön yli. Näin voitaisiin tarkkailla myös sovelluksen stabiliteettia, esimerkiksi muistivuotoja tai kirjautumissession katkaisemista tietyn ajanjakson jälkeen.

Tällä hetkellä regressiotestaus luottaa automaatiotestauksen testeihin, jotka ovat kyllä kattavia, mutta eivät aina välttämättä ajan tasalla. Automaatiotestit yleensä hieman laahaavat kehityksen jäljessä, eli ensin toteutetaan ominaisuus, sitten tehdään testit, jotka testaavat tätä. Mallipohjaisessa testauksessa voidaan tehdä malli samaan aikaan kuin itse sovellus, ellei jopa ennen sitä. Mallista luodut testit olisivat heti ajan tasalla, ja kehityksen aikana luodut testit voitaisiin siirtää luontevasti regressiotesteiksi kehityksen tuotantoon siirron jälkeen.

Mallipohjaisuus toisi apua myös dokumentaatioon. Kun sovelluksen toiminta olisi kuvattu malliin, olisi selkeästi kuvattu toiminta kaikkien testaajien apuna, jos jälkikäteen tarvitaan tietoa sovelluksesta. Tällä hetkellä dokumentointi on osittain vajavaista, eikä selkeää yhteisymmärrystä välttämättä ole kenelläkään, vaan jokaisella henkilöllä on oma ymmärrys sovelluksen toiminnasta kysyttäessä.

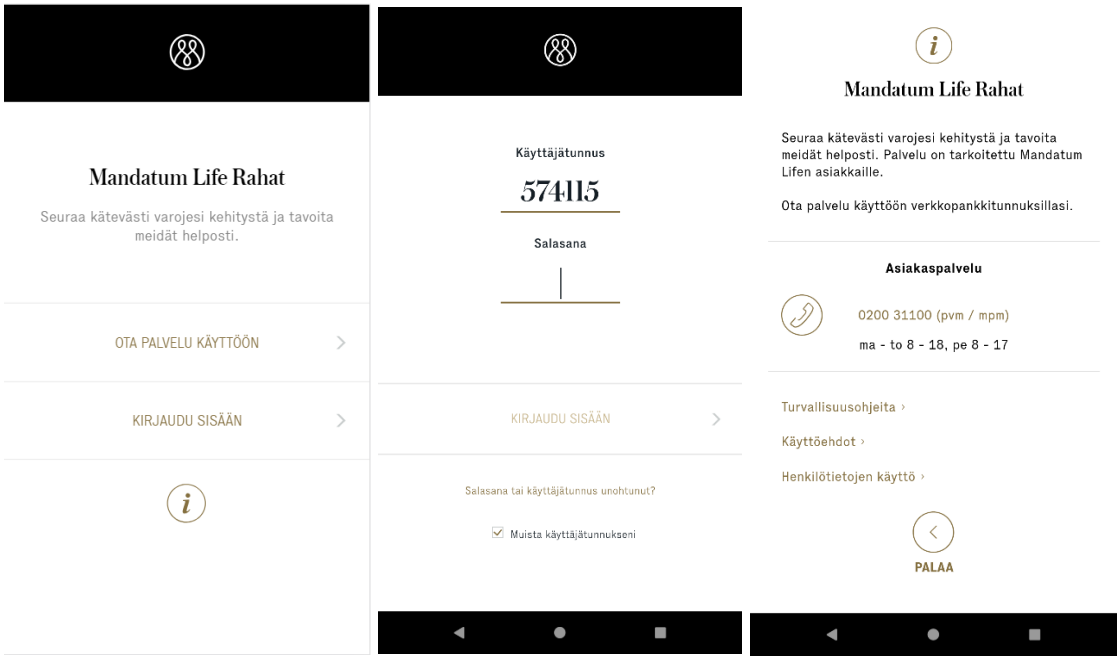
3. TESTATTAVAN SOVELLUKSEN ESITTELY

ML Rahat on Mandatum Lifen mobiilisovellus Mandatum Lifen asiakkaille, ja se on tarjolla sekä Android että iOS käyttöjärjestelmille. Sovellus tarjoaa mahdollisuuden asiakkaan omien sijoitusten tarkasteluun [9]. Asiakas voi sovelluksessa tarkastella sopimuskohtaisesti säästöjen kehitystä ja tuottoa. Yksi tällainen sopimus on vaurastumisen palvelun sopimus, mihin tässä työssä keskitytään. Lisäksi asiakas voi lähettää ja vastaanottaa viestejä asiakaspalvelusta, saada ilmoituksia viestejä asiakkuuteensa liittyen. Palvelu on saatavilla suomeksi, ruotsiksi ja englanniksi. Sovelluksen tarkoitus on antaa asiakkaalle väylä omiin sijoituksiinsa myös liikkeessä, ilman että tarvitsee kirjautua varsinaiseen verkkopalveluun. [9][10]

3.1 Sovelluksen esittely

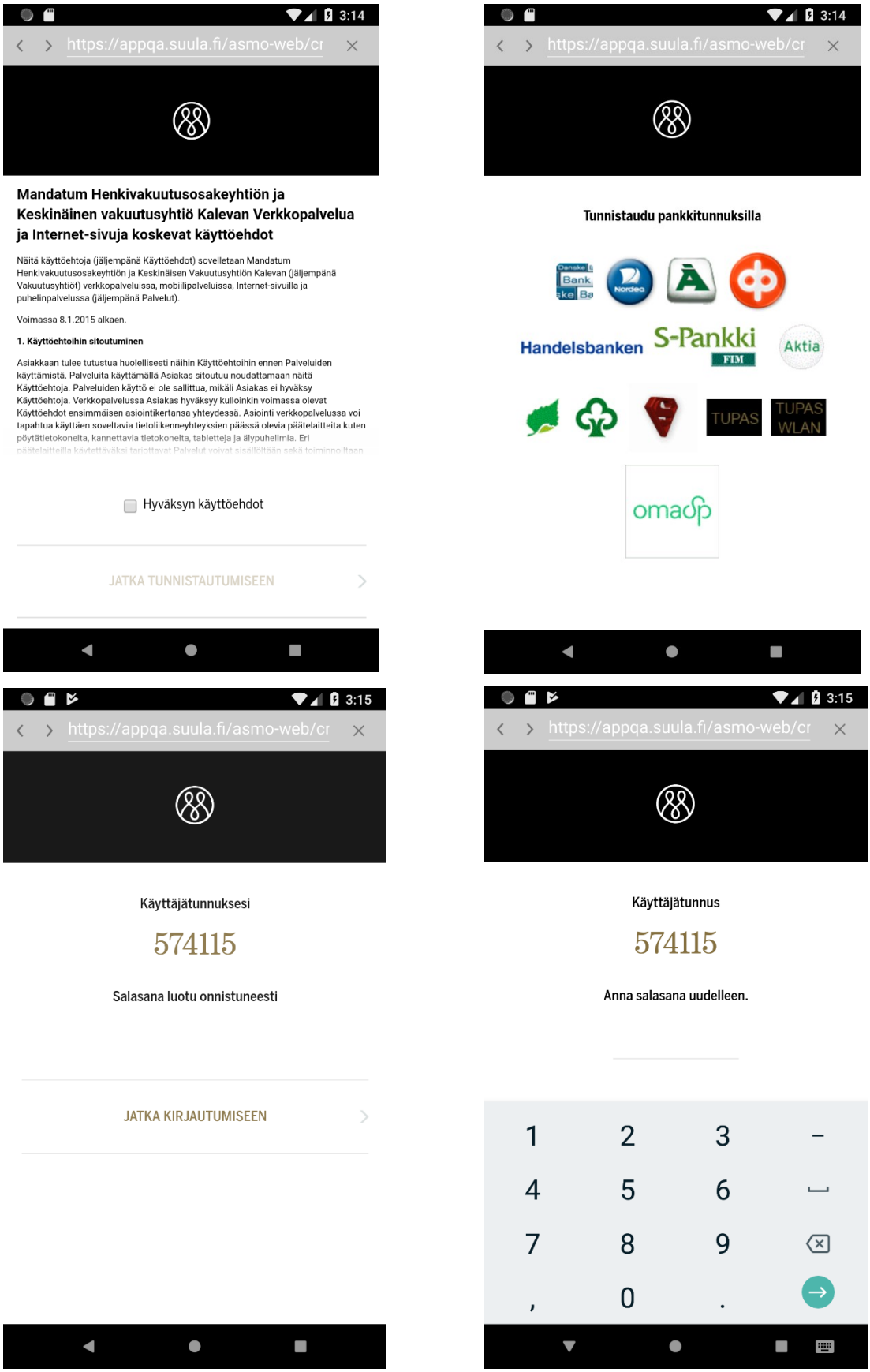
Tekniseltä toteutukseltaan ML Rahat on hybridiapplikaatio. Se tarkoittaa, että käyttöliittymä on webpohjainen, joka paketoidaan aputyökalun kuten Cordovan, avulla sovellukseksi joko Androidille tai iOSlle [11][12]. Hybridiapplikaatioiden etu on se, että molempia käyttöjärjestelmäversiota voidaan kehittää samanaikaisesti ja työkalu huolehtii, että sovellus toimii molemmissa käyttöjärjestelmissä. Molemmat versiot voidaan kehittää samanaikaisesti samasta koodipohjasta. Haittana on se, että kaikkea natiiviapplikaatioiden ominaisuuksia ei voida suoraan käyttää, vaan kaikkia toimintoja tulee käyttää aputyökalun, kuten Cordovan, tarjoamien rajapintojen kautta. Sovellukset evät välttämättä ole niin vakaat kuin natiiviapplikaatiot koska sovelluksissa on yksi ylimääräinen kerros sovelluksen ja laitteiston välissä, ja lisäksi hybridiapplikaation toiminta riippuu aina laitteeseen asennetusta selaimesta. [13]

ML Rahat -sovelluksen käynnistyessä avautuu käyttäjälle aloitussivu, jossa voi valita sisäänkirjautumisen tai rekisteröitymisen. Tässä onkin selkeä vedenjakajakohta sovelluksessa, eli mitä voi tehdä ilman kirjautumista ja mitkä ominaisuudet vaativat kirjautumisen. Kaikki päätoiminnot kuten sopimusten tarkkailu ja asiakaspalvelun kanssa kommunikointi vaatii sisäänkirjautumisen. Ilman kirjautumista pääsee käyttäjä näkemään sovellukseen liittyvät lakitekstit, kuten käyttöehdot sekä asiakaspalvelun numeron Infosivulta. Kuvassa 1 on nähtävillä aloitussivu, sisäänkirjautumissivu ja infosivu. Ilman sisäänkirjautumista on myös mahdollista luoda tunnus omilla verkkopankkitunnuksillaan eli rekisteröidä sovellus omaan käyttöönsä. Tunnusten luominen vaatii olemassa olevan asiakkuuden Mandatum Lifelle. Rekisteröintiprosessia on kuvattu kuvassa 2.



Kuva 1. *Aloitussivu, kirjautumissivu ja infosivu*

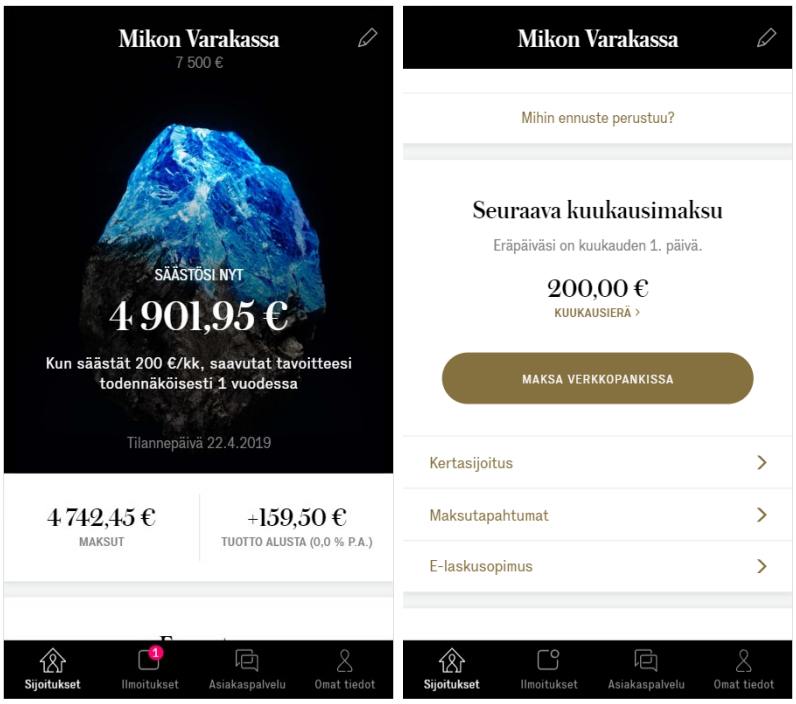
Sisäänkirjautumisen jälkeen käyttäjä ohjataan omien sopimustensa sivulle. Mikäli käyttäjällä on vain yksi sovelluksessa näytettävä sopimus, ohjataan käyttäjä suoraan tälle sopimussivulle, muutoin näytetään listaus käyttäjän kaikista sopimuksista. Käyttäjälle näytetään myös navigaatiopalkki laitteen alareunassa, jossa on neljä linkkiä sovelluksen keskeisiin toimintoihin. Nämä ovat sijoitukset, ilmoitukset, viestit ja omat tiedot. Nämä ovat sovelluksen tärkeimmät toiminnot ja jakavat sovelluksen neljään loogiseen kokonaisuuteen.



Kuva 2. *Rekisteröinti, pankin valinta, salasanan asetus ja tunnus luotu -sivu*

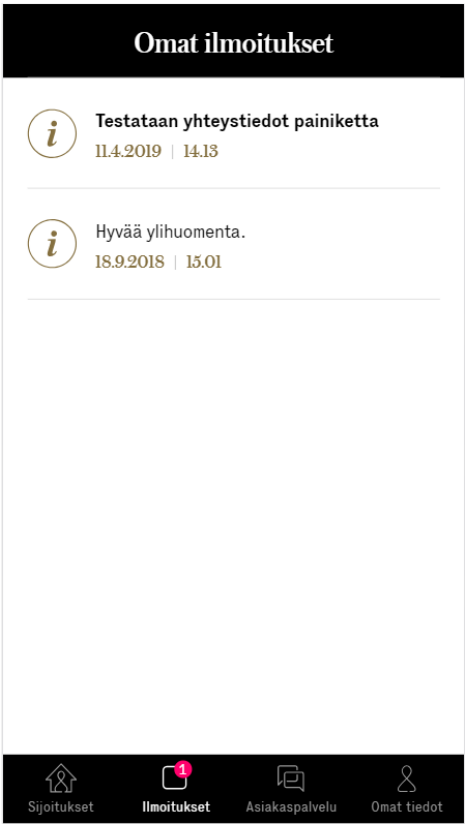
Sijoitukset sivuilla näytetään käyttäjän kaikki sopimukset. Jotakin sopimusta painettaessa, ohjataan käyttäjä kyseiselle sopimussivulle, jossa on yksityiskohtaista tietoa sopimuksesta. Jos käyttäjällä on vain yksi sopimus, ohjataan käyttäjä suoraan tälle sopimussivulle eikä hänelle näytetä listausta.

Tässä työssä keskitytään yhteen sopimukseen, joka on Vaurastumisen palvelun sopimus. Sopimus on tarkoitettu asiakkaille, jotka haluavat sijoittaa ilman jokapäiväistä huolehdintaa säästöistä [14]. Sopimuksessa asetetaan jokin tavoitesumma ja -aika, johon tähdätään. Sijoittamista voi vauhdittaa maksamalla joko satunaisia lisäsäästöjä sopimukseen tai maksamalla joka kuukausi tietyn suuruinen summa sopimukseen [8]. Kuvassa 3 on kuvattu Vaurastumisen palvelun sopimuksen etusivu sekä sopimukseen maksamisen linkit.



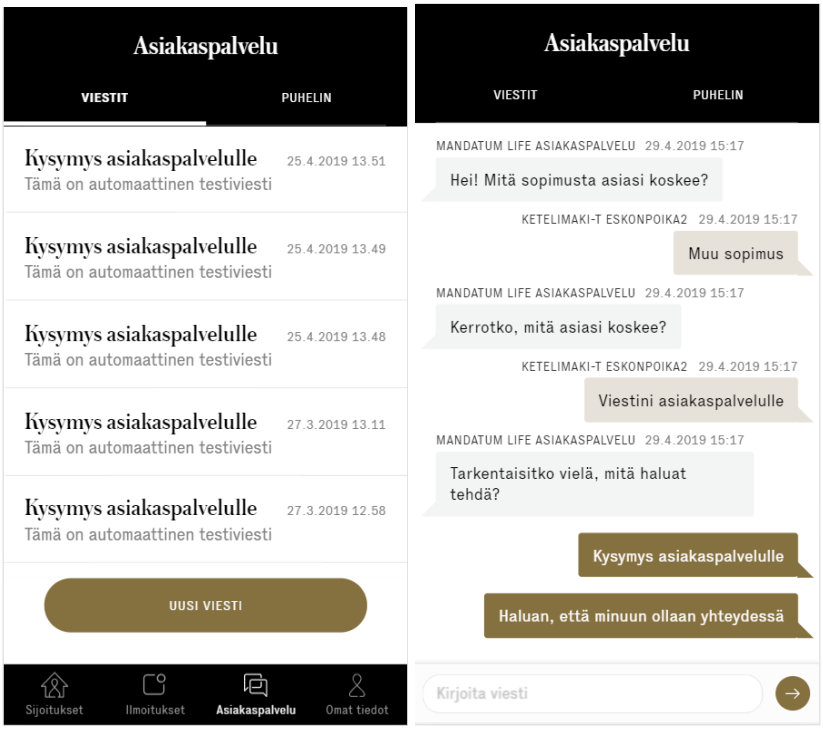
Kuva 3. *Vaurastumisen palvelun sopimuksen etusivu ja sopimukseen maksaminen*

Ilmoitukset sivuilla näytetään käyttäjän vastaanottamat ilmoitukset. Ilmoituksia tulee esimerkiksi, jos Mandatum Life lähettää yleisen tiedotteen, joka koskee käyttäjän sopimusta. Ilmoitusviesteistä tulee myös push-ilmoitusviesti, mikäli käyttäjä on ottanut ne käyttöönsä. Push-viestin avulla käyttäjä saa ilmoituksen uudesta viestistä, vaikka sovellus ei olisi käynnissä [15]. Jonkin ilmoituksen avaamisen ja sulkemisen jälkeen kysytään käyttäjältä, haluaako hän ottaa push-viestit käyttöön. Tämä kysytään joka kerta, kunnes käyttäjä ottaa push-viestit käyttöön. Ilmoitukset sivu on kuvattu kuvassa 4.



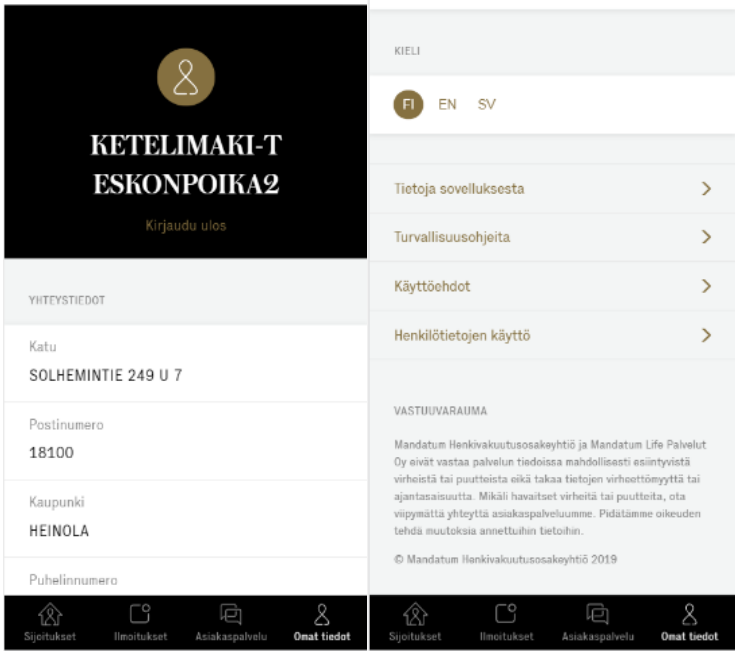
Kuva 4. *Ilmoitukset sivu*

Asiakaspalvelusivulta käyttäjä voi lähettää viestejä asiakaspalveluun. Myös hänen asiakaspalvelustaan vastaanottamansa viestit näkyvät sivulla. Viestit näkyvät viestiketjuina, jotta kommunikointi asiakaspalvelun kanssa olisi mahdollisimman sulavaa. Lähettäessään uutta viestiä saa käyttäjä valittavaksi aiheen, mitä asiaa viesti koskee. Viestin aihe voi koskea jotain käyttäjän sopimusta tai olla yleinen kysymys asiakaspalvelulle. Viestien etusivu ja uuden viestiketjun aloittaminen on kuvattu kuvassa 5.



Kuva 5. *Asiakaspalvelu -sivu ja uuden viestin luominen*

Omat tiedot -sivulta käyttäjä näkee omat tietonsa, kuten osoitteen ja nimen. Tietoja ei voi muuttaa tällä sivulla. Sivulla on myös kielivalinta, josta kielen voi vaihtaa suomeksi, ruotsiksi tai englanniksi. Sivun alareunassa on osittain samat lakitekstit kuin sisäänkirjautumisen infosivulla. Omat tiedot -sivu ja kielen valinta sekä lakitekstien linkit näkyvät kuvassa 6.



Kuva 6. *Omat tiedot ja kielenvalinta*

3.2 Nykytilanne sovelluksen testauksessa

Sovellusta testataan tällä hetkellä sekä manuaalisesti että automaattisesti. Uutta kehitystä tehdään Scaled Agile Framework (SAFe) mallin mukaan, jossa jokin kehitystiimi toteuttaa uusia ominaisuuksia sovellukseen, joita tiimin nimetty testaaja sitten testaa [14]. Tiimin testaaja testaa tällä hetkellä pääosin manuaalisesti uutta kehitystä, jonka tueksi on automaatiotestaus regressiotestauksen roolissa.

3.2.1 Manuaalitestaus

Manuaalitestausta tehdään loppukäyttäjän roolissa suoraan sovelluksen käyttöliittymän kautta ja se keskittyy uusien ominaisuuksien testaamisen. Testaus on pyritty tuomaan mahdollisimman lähelle kehittämistä, testaajat kuuluvat osaksi kehitystiimiä ja vastaavat tiimin toteuttamien ominaisuuksien testauksesta. Testitapaukset luodaan määritelmien perusteella sekä testaajien omaa kokemusta ja ammattitietoa hyödyntäen. Usein määritelmät ovat kuitenkin puutteellisia, jolloin tulee vastaan tilanteita, joissa sovellus toimii epätavanomaisesti, mutta tilannetta ei voi suoraan todeta häiriöksi. Talloin on varmistettava sovelluksen toiminnan oikeellisuus joko Product Ownerilta tai Epic Ownerilta, jotka omistavat ja ovat määritelleet sovelluksen toiminnan [17][18].

Tiimien töitä ja vastuualueita ei ole ennalta määritelty kuin yhdeksi toteussykliksi, joten välttämättä seuraavissa sykleissä sama tiimi ei työskentele saman sovelluksen parissa, joten myöskin eri testaajat testaavat välillä eri sovelluksia. Koska määritelmät ovat puutteellisia, eivät testaajat voi tutustua sovellukseen ennen uusien toiminnallisuuksien testausta muuten kuin katsomalla, miten sovellus toimii. Manuaalitestauksen ongelmia ovat tällä hetkellä siis puutteelliset määritelmät, joiden takia testaajat eivät saa kunnollista käsitystä sovelluksen toiminnasta.

3.2.2 Testausautomaatio

Automaatiotestausta hyödynnetään sovelluksen testaamisessa regressiotestauksen muodossa. Automaatiotestit ajetaan kaksi kertaa päivässä sekä tuotantoasennuksien yhteydessä useammin. Automaatiotestauksen testitapaukset toteutetaan manuaalitestauksen testitapausten pohjalta, kun uusi toiminnollisuus on testattu manuaalisesti ja se on menossa tuotantoon. Sopivat manuaaliset testitapaukset automatisoidaan Robot Frameworkillä suoritettavaksi. Näin tapauksia voidaan ajaa regressiotestauksena jatkossa, kun tulee uusia toiminnollisuuksia, jotka voivat vaikuttaa vanhaan toiminnollisuuteen.

Automaatiotestauksesta aikana löytyvät virheet tulevat usein liian myöhään kehityksen kannalta. Mitä aikaisemmin kehitysvaiheessa häiriöt löytyvät, sitä helpompi on korjata

ne, kun asiat ovat vielä tuoreessa muistissa. Siksi myös automaatiotestaukseen tulisi päästä mukaan mahdollisimman aikaisessa kehityksen vaiheessa.

Automaatiotestauksen nykytilanne on hyvä, mutta sitä voisi hyödyntää enemmän ja myös muussa testauksessa kuin pelkästään regressiotestauksessa. Automaatiotestaus on tehokasta, ja kun sitä käytetään yhdessä manuaalitestauksen kanssa, testauksen tehokkuus nousee [5]. Nykyisen automaation ongelma on, että testaajan tarvitsee erikseen luoda testitapaukset manuaalitestauksen pohjilta, ja usein eri testaajan toimesta. Jos automaatiotestien tekijä ei ole perehtynyt syvällisesti sovelluksen toimintaan, voivat testit jäädä vajavaiseksi, varsinkin jos manuaalitestauksen suorittanut henkilö ei perehdytä automaatiotestien luojaa tarpeeksi.

3.3 Mitä on tarkoitus tehdä projektin kannalta

Tässä työssä on tarkoitus luoda mallipohjainen automaatiotestaus ML Rahat -sovellukselle. Tarkoitus on kuvata kirjautuminen, sovelluksen runko ja yksi sopimus siten, että käytännössä käyttäjän, jolla on vain yksi sopimus, tilanne saadaan mallinnettua kokonaisuudessaan. Mallipohjainen testaus voisi auttaa sekä mobiilisovelluksen regressiotestauksessa että uuden kehityksen testauksessa.

Nykyiset automaatiotestit ajavat samoja testejä läpi, joiden oletusarvo on, että ne menevät läpi onnistuneesti kuten ennenkin. Ne testaavat siis vain samoja asioita joka kerta. Kun testit ovat löytäneet virheet ja ne ovat korjattu, niin nämä testit eivät enää löydä uusia virheitä, virheet tulevat ikään kuin immuuneiksi näille testeille. Tämä on regressiotestauksen pääidea, mutta se voi antaa myös väärän vaikutelman, että sovellus on täysin toimiva, kun testit suoritetaan aina läpi onnistuneesti, vaikka ne eivät kata kaikkia sovelluksen toimintoja. Sovelluksessa voi olla muualla häiriöitä, jotka eivät ole ilmenneet testauksessa uuden kehityksen aikana. Kun jotain uutta on kehitetty, vaikuttaa se kokemuksen mukaan ainakin välillisesti vanhaan toiminnallisuuteen, voiden aiheuttaa häiriön.

Mallipohjaisessa testauksessa testitapaukset luodaan automaattisesti työkalulla. Testitapaukset ovat tällöin aina halutun laisia eli sen mukaisia, mitä kriteereitä testejä luova työkalulle antaa. Mallipohjaisuudella saadaan siis luotua kattavasti monenlaisia testitapauksia. Sitä voidaan käyttää myös regressiotestauksessa luomalla yksi kattava testitapaus, jota käytetään toistuvasti ja ajetaan aina, kun sovellus päivittyy. Mallipohjaisessa testauksessa voidaan luoda myös uusia testitapauksia, jotka testaavat sovellusta yleisemmin.

Tällä hetkellä ei ole helppoa tapaa käydä koko ML Rahat -sovellusta läpi. Nykyiset automaatiotestit kattavat vain osan koko sovelluksen toiminnollisuudesta. Mallipohjaisella

testauksella voisi käydä sovelluksen ainakin teoriassa kokonaisuudessaan läpi, mikä mahdollistaa monenlaisten eri testitapauksien luomisen.

Mallipohjainen testaus auttaisi myös suorituskykytestauksessa. Luomalla erittäin pitkä testitapaus, voisi testin laittaa esimerkiksi yöksi tai viikonlopuksi ajoon, ja katsoa onko mobiilisovellus toimintakykyinen koko testin ajan. Sovellusta voidaan tarkkailla esimerkiksi muistivuotojen varalta. Tietty testitapaus voidaan tallentaa ja käyttää sitä ajamaan uudelleen sama testi, jolla voitaisiin mitata testiin kulunutta aikaa ja vaihteleeko se esimerkiksi eri käyttäjien välillä, joilla on erilaisia sopimuksia.

Tämän hetkistä tilannetta voisi siis mahdollisesti parantaa mallipohjaisuuden avulla. Kun nyt automaattitestaus on mukana ainoastaan regressiotestauksena, voisi mallipohjaisuuden avulla päästä käyttämään automaattitestausta heti kehityksen ja testauksen alkuvaiheessa sekä tuoda lisää testikattavuutta regressiotestaukseen.

3.4 Tulosten seuranta ja vertaaminen

Työn tarkoituksena on toteuttaa sovellukseen malli, kattaen kirjautumisen, sovelluksen yleiset toiminnot sekä yhden sopimuksen. Mallin pohjalta luodaan ja ajetaan testejä ja analysoidaan tuloksia verraten työmäärää sekä testien kattavuutta vanhoihin automaatiotesteihin. Jos tulokset osoittavat, että mallipohjainen testaus olisi kannattavaa, niin tämän työn pohjalta voidaan mallia laajentaa kattamaan koko sovellus sekä mahdollisesti käyttää apuna uusien sovelluksen kehittämisessä, jos niihin halutaan ottaa mallipohjainen prosessi mukaan.

Löytyneitä häiriöitä voisi myös käyttää yhtenä kriteerinä arvioimaan mallipohjaista testausta, mutta häiriöiden määrä on yleensä huono kriteeri [3]. Muutenkin ML Rahat on jo asiakkailla käytössä oleva sovellus, ja kun työssä ei testauksen kohteena ole varsinaista uutta kehitystä, niin oletuksena on, että löytyneet häiriöt eivät ole kriittisiä tai että häiriöitä ei löydy kovin monta.

Löytyneet häiriöt havaitaan siitä, että mallin pohjalta luodun testin suorittaminen pysähtyy. Tässä vaiheessa tämä on käypä ratkaisu, mutta jatkossa voisi olla järkevää erotella häiriöt kriittisiin ja ei-kriittisiin. Esimerkiksi voisi merkata häiriöt ei-kriittisiksi niin kauan kun pystytään suorittamaan seuraava askel testistä, eli niin kauan kuin seuraava siirtymä on mahdollinen, voitaisiin testin suorittamista jatkaa. Nyt pitkäkin testi voi pysähtyä jo alkuvaiheessa, jos testin aikana törmätään johonkin häiriöön.

Toinen ratkaisu voisi olla merkata estetyksi se tila tai siirtymä missä törmätään häiriöön, minkä jälkeen luodaan mallista uusi polku. Tässä metodissa saataisiin suoritettua mah-

dollisimman paljon testejä mallista ja vain ne testipolut, jossa törmätään ongelmiin, jäisivät suorittamatta. Estettyjä tiloja ja siirtymiä voisi sitten tarkastaa siten että testaaja seuraisi testien suoritusta katsoen millaisia häiriöitä testeistä ilmenee, ovatko ne oikeita häiriöitä vai mallin virheitä tai onko sovellus muuttunut taustalla ja malli vaatii päivittämistä.

4. MALLIN LUOMINEN PROJEKTIIN

Mallipohjainen testaus vaatii ensin mallin, joka nyt toteutetaan olemassa olevan järjestelmän nykyisen toiminnan perusteella. Kuten mainittua, parempi testausmalli saataisiin, jos malli toteutettaisiin puhtaasti määrittelyiden pohjilta. Määrittelyiden ollessa rajallisia tulee malli toteuttaa sovelluksen nykyisen toiminnan pohjita. Tämä on myös ihan hyvä toimintatapa, mutta luotu malli luottaa nyt siihen, että sovellus sen perustana toimii oikein. Tällöin on riskinä, että mahdolliset häiriöt tulevat malliin mukaan ominaisuuksina, kun luullaan että häiriöt ovat haluttua toimintaa [5].

4.1 Mallinnuksen työnkulku

Mallinnusprosessi aloitetaan asentamalla tarvittavat työkalut. Tämän jälkeen voidaan alkaa tutustumaan sovelluksen toimintaan käymällä sitä läpi ja lukemalla saatavilla olevaa dokumentaatiota. Samalla aletaan hahmottelemaan sovelluksen toimintaa malliksi. Mallissa nuolet kuvaavat siirtymiä ja laatikot kuvaavat tiloja. Siirtymät ovat toimenpiteitä, joita tarvitsee tehdä, jotta päästään yhdestä tilasta toiseen.

Kun malli on saatu luotua, luodaan mallipohjaisen testauksen työkalulla mallin pohjilta testitapauksia, testipolkuja. Samalla tulee varmistettua, että malli on oikeanlainen, eli sen perusteella saadaan luotua testitapauksia eikä saada virheilmoituksia, ja jokainen tila ja siirtymä on saavutettavissa. Luotujen polkujen pohjalta toteutetaan tarvittavat testit, eli jollain testauskehyksellä toteutetaan testipolun siirtymät ja tilat. Tässä työssä testit toteutetaan Robot Frameworkillä käyttäen apuna Appiumia. Lopuksi voidaan ajaa testipolkua testien kanssa, jolloin suoritetaan itse mallipohjaista testausta.

4.2 Käytettävät työkalut

Työssä käytetään pääosin viittä työkalua, jotka ovat yEd, Graphwalker, Robot Framework ja Appium. Käytetyt työkalut valintaperusteineen esitellään seuraavaksi.

4.2.1 yEd

yEd on ilmainen ohjelma graafien piirtämiseen. Se on tarjolla Linuxille, Mac OS:lle ja Windowsille. Windows-alusta on yleisesti käytössä Mandatum Lifellä, joten valittujen ohjelmien on toimittava Microsoftin ympäristöissä. yEd tarjoaa tuen monen erityyppisen graafin piirtämiseen, kuten UML, vuokaaviot tai verkostodiagrammit. yEd tallentaa dia-

grammit GraphML muotoon [19]. yEd toimii hyvin yhteen valitun mallipohjaisen testauksen työkalun, Graphwalkerin kanssa. Tämä yhdistettynä sen ilmaisuuteen ja Windows tukeen tekee siitä luontevan valinnan graafien piirtämiseen.

4.2.2 Graphwalker

Graphwalker on mallipohjaisen testauksen työkalu. Se lukee sille annettua malleja suunnattuina graafeina, ja luo testipolkuja eli -tapauksia mallista kulkemalla sen läpi jonkin valitun algoritmin mukaan. Graphwalker on ilmainen ja open-source sovellus. Se on Kristian Karlin kehittämä ja ylläpitämä. Karl on töissä Spotifylla testauksen parissa ja Graphwalker onkin käytössä Spotifyllä mallipohjaisessa testauksessa. [20]

Graphwalker omaa paljon dokumentaatiota sen käytöstä ja toiminnasta sen verkkosivuilla. Lisäksi Karl ja muutama muu henkilö kehittävät ja ylläpitävät aktiivisesti Graphwalkeria [21]. Graphwalkerilla on myös aktiivinen Google Groups -foorumi, jossa sen kehittäjät vastailevat kysymyksiin.

Graphwalker tukee myös Windows alustaa, koska se on alusta riippumaton. Graphwalker vaikuttaa lupaavalta mallipohjaisen testauksen työkalulta, joka on käytössä myös muualla oikeissa projekteissa. Tämä yhdistettynä sen aktiiviseen kehittämiseen ja tukeen, saa sen vaikuttamaan sopivalta työkalulta mallipohjaisen testauksen toteuttamiseen.

Muita tutkittuja työkaluja ovat muun muassa fMBT, joka kuitenkin tukee vain Linux-alustaa ja RoboMachine, joka olisi ollut suoraan Robot Frameworkille tarkoitettu työkalu, mutta sen kehitys vaikuttaa lakanneen [22][23].

4.2.3 Robot Framework

Robot Framework on Python pohjainen open-source automaatiotestikehys, joka käyttää avainsanapohjaista rakennetta testien kuvaamiseen. Robot Framework sopii erityisesti hyväksymistestaukseen. Mandatum Lifen automaatiotestit on toteutettu Robot Frameworkillä. Testejä ajetaan regressiotesteinä testaamaan vanhan toiminnallisuuden toimivuus, kun uutta kehitystä integroidaan vanhan kehityksen pohjalle. Robot Frameworkin avulla voidaan testata automaattisesti sekä selaimia että mobiilisovelluksia [24].

Robot Frameworkissä jokin avainsana kuvaa toiminnon, joka suoritetaan sitten testauksen kohteena olevassa ohjelmistossa. Mallipohjaisessa testauksessa tätä voidaan hyödyntää toteuttamalla mallin siirtymät ja tilat Robot Frameworkillä kuvaamalla ne avainsanoiksi. Näin mallista ja sen testeistä tulee helpommin ymmärrettävä myös muille kuin testaajille verrattuna siihen, että mallin testit toteuttaisin suoraan testattavan sovelluksen

koodilla. Tällöin testausvastuu siirtyisi lähemmäksi kehitystä ja kehittäjän vastuulle testaajilta, sillä testaajilla ei välttämättä ole niin tekninen tausta, että tämä olisi mahdollista.

Robot Framework on aktiivisessa käytössä tämän hetken testiautomaatiossa, joten on luontevaan lähteä toteuttamaan myös mallipohjaista testausta tällä tekniikalla, vaikka sitä ei suoraan olekaan tuettu. Näin mallipohjainen testaus saadaan integroitua olemassa olevaan regressiotestauksen avuksi, eikä korvaamaan täysin olemassa olevaa regressiotestausta. Lisäksi päästää hyödyntämään jo tehtyjä testejä, käyttämään niitä uudelleen tai hyödyntämään valmiita apukirjastoja, joita eri sovelluksille on jo tehty.

4.2.4 Appium

Appium on open-source ohjelma mobiilisovellusten testaamisen. Se mahdollistaa mobiilisovellusten ohjaamisen automaattisesti Webdriverin avulla, eli samanlailla kuin Selenium mahdollistaa selainten ohjauksen. Appiumista on saatavilla valmis kirjasto Robot Frameworkille, mikä mahdollistaa Robot Frameworkin käytön myös mobiilisovellusten testauksessa. [25][26]

Appium on valittu vuonna 2014 parhaaksi open-source ohjelmaksi. Appium on myös nykyisin käytössä Mandatum Lifella mobiilisovellusten automaatiotestauksessa, joten on luonnollista käyttää sitä myös tässä työssä.

4.2.5 Koodieditori

Robot Frameworkin testien kirjoittamista varten tarvitaan jokin koodieditorin, joka tukee Robot testien syntaksia. Käytännössä mikä tahansa editori käy, tässä työssä testit on kirjoitettu Microsoftin Visual Studio Code -editorilla, johon on saatavana Robot Framework plugin sen syntaksia varten. Visual Studio Code on saatavilla vapaasti sen verkkosivuilta. Muita vaihtoehtoisia editoreja voisivat olla Robot Frameworkin oma RIDE editori, PyCharm tai vaikkapa Notepad++.

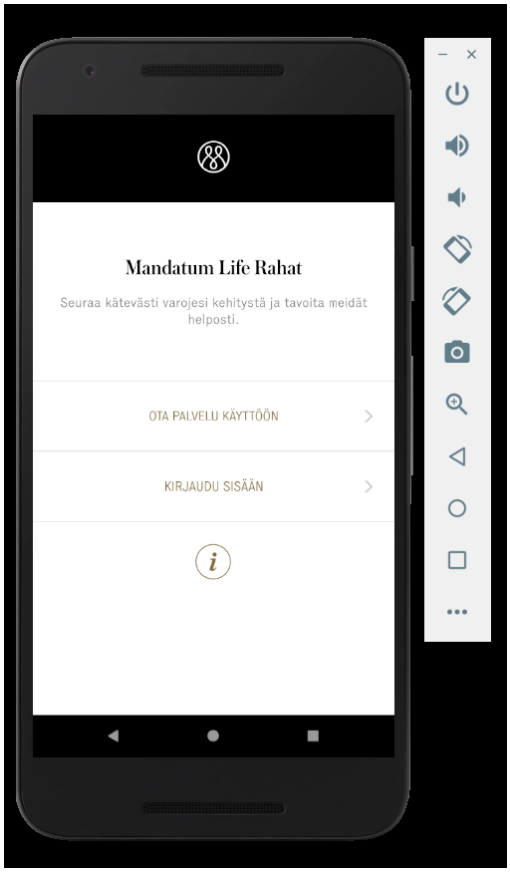
4.3 Kehitysympäristön pystytys

Kehitysympäristö on pystytettävä, jotta mallipohjaista testausta voidaan ensin suunnitella ja lopulta suorittaa. Käytettävät työkalut asennetaan sitä mukaan, kun niitä tarvitaan tai kaikki työkalut voi asentaa kerralla. Kappaleessa 4.2 esitellyt työkalut tulee asentaa kehitysympäristöön mahdollisten riippuvuuksien kera. Tässä työssä työkalujen asennus Windows-alustalle tapahtuivat pääosin käyttäen asennuspaketteja työkalujen kotisivuilta ja noudattamalla niiden ohjeita.

Graphwalker tulee asentaa lataamalla koneelle valmiiksi käännetty binääri Grahpwalkerin kotisivuilta. Näin vältytään turhien apuohjelmien asennukselta ja varmistetaan että työkalusta käytettävä versio on kaikilla käyttäjillä sama, kun käytetään viimeisintä stabiilia versiota. [20]

Robot Frameworkin asennus vaatii Pythonin asennuksen. Robot Framework ja tarvittavat kirjastot selainten ja mobiilisovellusten automaattiseen ohjaamiseen saadaan asennettua Pythonin pakettimanagerilla, PIP:llä. Ainakin Appium-apukirjasto on asennettava, jotta mobiilisovellusta saadaan ohjattua testeillä. [24]

Lopuksi tarvitaan vielä laite, jossa on ML Rahat -sovellus asennettuna. Voidaan joko käyttää fyysistä laitetta tai emuloida laitetta Googlen Android Studio ja Android Virtual Device Managerin (AVD) avulla. AVD:n avulla saadaan suoraan tietokoneessa ajettua emuloitua versiota koko Android-käyttöjärjestelmästä. Tässä työssä käytetään Android emulaattoria ML Rahat -sovelluksen ajamiseen. Kuvassa 7 on nähtävillä ML Rahat -sovellus ajettuna emulaattorilla.

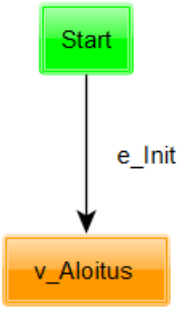


Kuva 7. *ML Rahat -sovellus emulaattorissa*

4.4 Sovelluksen mallien luominen ja tekeminen

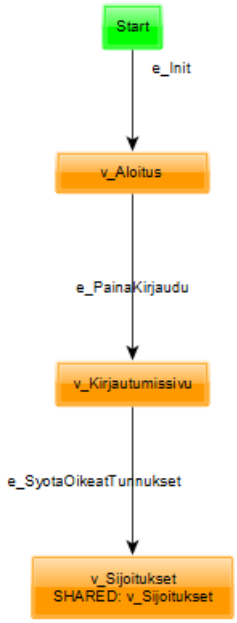
Ensimmäinen askel mallin luomiseen on aloittaa piirtämään mallia yEd ohjelman avulla. Mallissa siirtymät ovat kaaria ja tilat solmuja. Kaareen liittyy toimenpiteitä, joita tarvitsee tehdä, jotta päästään yhdestä tilasta toiseen. Kaaret ovat siis usein jotain mitä käyttäjä tekee, esimerkiksi painaa nappia. Solmuihin on liitetty tarkistuksia, jotka varmistavat, että oikea tila on saavutettu, esimerkiksi että ollaan oikealla sivulla. Selvyyden vuoksi nimetään kaaret alkamaan e-kirjaimella ja solmut v-kirjaimella.

Aloitetaan aluksi siitä, miten sovellus saadaan käyntiin. Graphwalker vaatii ensimmäisen solmun nimen olevan ”Start”, joten piirretään se graafiin. Seuraava tila on se, miltä sovellus näyttää, kun käyttäjä on käynnistänyt sen. ML Rahoissa näytetään käyttäjälle ensin aloitussivu, joten nimetään tämä tila v_Aloitukseksi. Ensimmäinen kaari on näiden kahden tilan välissä. Tässä tilassa käynnistetään sovellus emulaattorissa ja valmistellaan se vastaanottamaan Appiumin avulla Robot Framworkin komentoja. Tätä siirtymää voidaan kuvata nimellä e_Init. Kuvassa 8 on kuvattu tilat Start ja v_Aloitus sekä niiden välinen kaari e_Init.



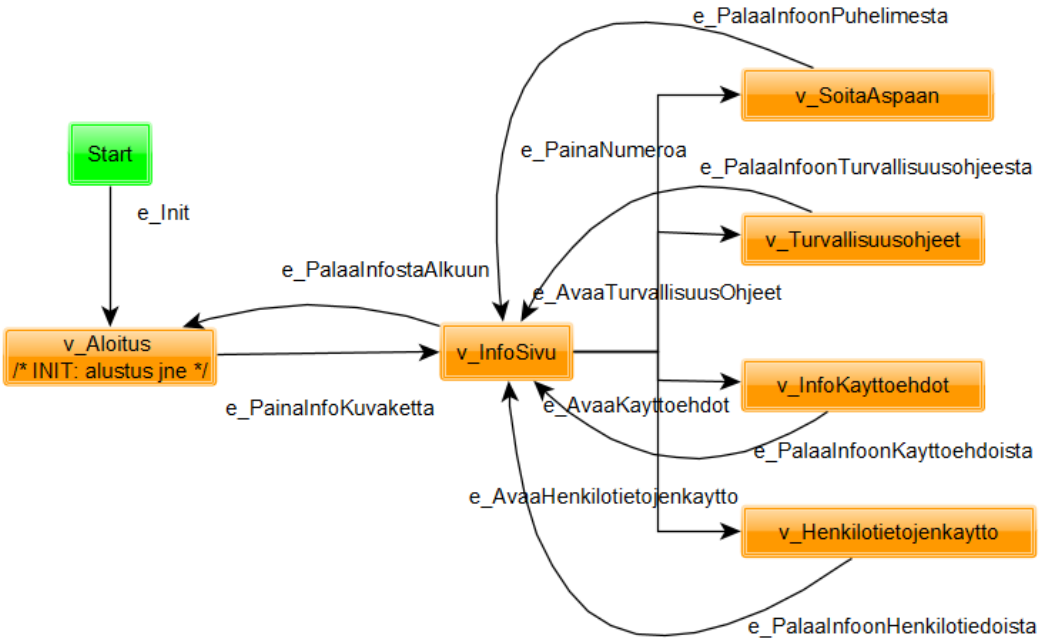
Kuva 8. *Ensimmäinen siirtymä ja tila yEdillä piirrettynä*

Kun sovelluksen käynnistyminen on saatu mallinnettua, voidaan alkaa mallintamaan sovelluksen muita tiloja ja siirtymiä. ML Rahat -sovellus vaatii aina kirjautumisen käynnistyessään, joten on luontevaa mallintaa ensin kirjautumistapahtuma. Mallinnetaan ensin siis validi kirjautuminen, josta muodostuukin suora putki kirjautumisvaiheen läpi. Kirjautumisputki on kuvattu kuvaan 9. Kirjautumisen mallintamisen jälkeen palataan aloitussivulle ja katsotaan läpi muut siirtymät, joita v_Aloitus tilasta on mahdollista tehdä. Solmusta voi aloittaa rekisteröintiprosessin sekä päästä infosivulle.



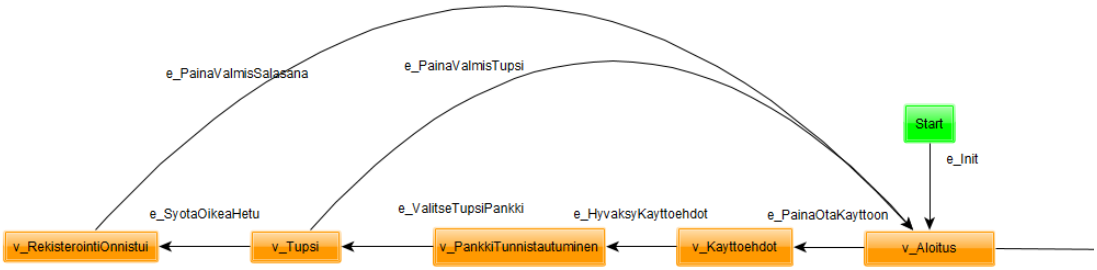
Kuva 9. *Kirjautuminen mallinnettuna yEdillä*

Infosivu on suhteellisen yksinkertainen. Se sisältää linkit turvallisuus- ja käyttöehtoihin sekä henkilötietojen käyttöön, jonka lisäksi sivu sisältää myös asiakaspalvelun numeron. Linkkejä klikattaessa avataan sovelluksen päälle verkkoselainikkuna sovelluksen sisässä, niin sanottu In-App Browser (IAB). Tämä aiheuttaa todennäköisesti joitain ongelmia, kun näille siirtymille toteutetaan robottitestit, sillä tavallaan siirrytään pois ML Rahat -sovelluksesta verkkoselaimeen. Kuitenkin toistaiseksi voidaan keskittyä pelkästään mallin piirtämiseen yEdillä. Asiakaspalvelun numeroa painettaessa avataan laitteen puhelinsovellus, josta käyttäjä pääsee mahdollisimman helposti soittamaan asiakaspalveluun. Myös tämä voi aiheuttaa hankaluuksia, sillä tässä siirtymässä siirrytään kokonaan eri sovellukseen. Infosivun mallinnusta on kuvattu kuvassa 10.



Kuva 10. *Infosivun ja sen linkkien mallinnus yEdissä*

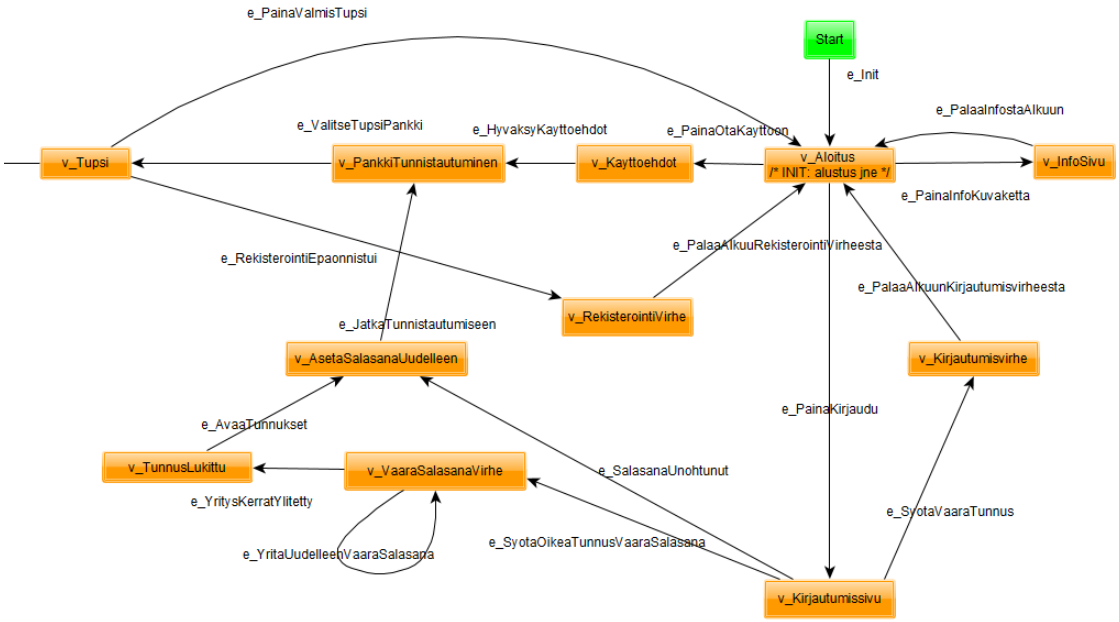
Infosivua kiinnostavampi polku on tunnuksen luonti sovellukseen eli rekisteröinti, joka tapahtuu suurimmaksi osaksi IAB:ssa. Polun mallinnuksessa ei sinänsä ole mitään vaikeaa tai erityistä, mutta polku on suhteellisen pitkä ja se tulee mallintaa huolellisesti. Iso asia mikä tulee ottaa huomioon, on, että IAB on suljettavissa koko polun ajan missä tilassa vain. Jos tämän mallintaa tarkasti joka vaiheen osalta, tulee mahdollisten testitapausten määrä kasvamaan paljon. Kannattaakin sen sijaan keskittyä mallintamaan testauksen näkökulmasta kiinnostavimmat kohdat selainikkunan sulkemiselle [3][5]. Näitä on esimerkiksi IAB:n sulkeminen juuri tunnuksen luomisen jälkeen ennen salasanan asettamista tai ennen kuin salasana on varmistettu. Kuvassa 11 on kuvattu rekisteröintipolkua. Myös tiloissa v_Kayttoehdot ja v_Pankkitunnistautumien on mahdollista sulkea IAB, mutta ne eivät ole testauksen kannalta merkittäviä tapauksia.



Kuva 11. *Rekisteröinnin polku piirrettynä yEdissä*

Testauksen kannalta merkittävät tilanteet ovat tilanne, jossa käyttäjälle on luotu tunnus, mutta sille ei ole asetettu salasanaa, eli tila v_RekisterointiOnnistui. Jos tässä kohdassa sulkee IABn, pääseekö käyttäjä ilman salasanaa kirjautumaan sisään sovelluksen, onko salasanan arvo tyhjä, ja miten se vaikuttaa sovelluksen käyttäytymiseen? Jos käyttäjä tunnistautuu uudelleen, tulisi hänen saada normaalisti asetettua salasanansa, mutta vaikuttaako puuttuva salasana sovelluksen toimintaan tässä tilanteessa jotenkin? Nämä ovat hyviä testitapauksia, jotka on syytä kuvata malliin mahdollisiksi poluiksi.

Näiden tilojen ja siirtymien jälkeen on sovelluksen kirjautumispuoli pääpiirteittäin kuvattu malliin ja malli alkaa olemaan koko kirjautumisprosessin kattava. Lopuksi tulee kuitenkin muistaa kuvata malliin mahdollisia virhetilanteita. Validin kirjautumisen vastakohdiksi tulee kuvata myös tilanteet, joissa yritetään kirjautua väärällä tunnuksella sisään, ja tilanne, jossa yritetään kirjautua oikealla tunnuksella mutta väärällä salasanalla. Tässä tilanteessa kolmen virheellisen kirjautumisyrityksen jälkeen tulee käyttäjän tunnuksen lukittautua. Käyttäjä saa avattua tunnuksensa pankkitunnistautumisella. Myöskin virheellisellä henkilötunnuksella rekisteröinti on kuvattava sekä myös tilanne, jossa käyttäjä keskeyttää rekisteröinnin pankkitunnistautumisen aikana. Näitä virhetilanteita on kuvattu mallin kuvassa 12.



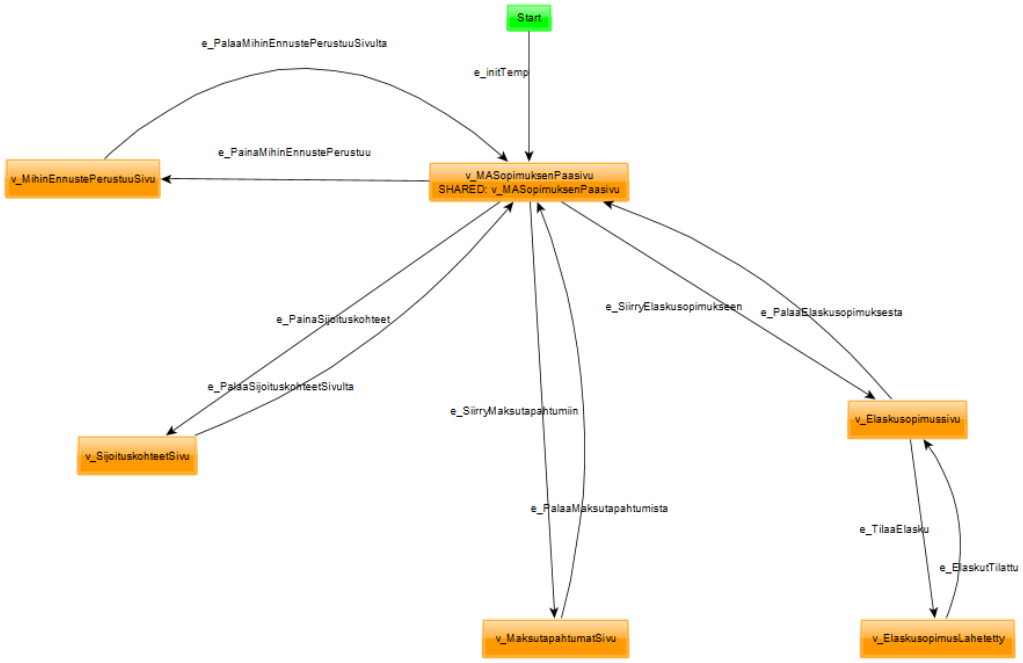
Kuva 12. *Virhetilanteita kuvattuina yEdissä*

Nyt on koko kirjautumisprosessi kuvattu ensimmäiseen malliin. Mallin oikeellisuuden voi varmistaa luomalla Graphwalkerilla testipolkuja. Mikäli saadaan luotua muutama polku eri ehdoilla, voidaan päätellä, että malli on validi. Kirjautumisprosessi on oma looginen kokonaisuutensa, joten seuraavat ominaisuudet aloitetaan kuvaamalla ne omaan malliinsa. Kirjautumisprosessin malli on kokonaisuudessaan liitteessä A.

Kirjautumisen jälkeen käyttäjä pääsee käyttämään sovelluksen avaintoimintoja. Sisään kirjautuneena käyttäjälle näkyy alanavigaatiopalkki, joka kuvaa sovelluksen eri osa-alueet: sijoitukset, ilmoitukset, asiakaspalvelun ja omat tiedot. Nämä ovat kirjautumisprosessin tapaan omia itsenäisiä kokonaisuuksiaan, mutta eivät kuitenkaan niin isoja kokonaisuuksia, että ne kaikki vaatisivat oman mallinsa. Sijoitukset välilehdestä eli käyttäjän sopimuksista kuvataan oma mallinsa, mutta muut osa-alueet voidaan kuvata kaikki samaan malliin. Näin voidaan myöhemmin laajentaa kokonaisuutta tekemällä eri sopimuksista omat mallinsa, jotka yhdistetään toisiinsa. Jos pieniä malleja on monta, niin kokonaiskuvasta ei välttämättä tule selkeää ja helposti ymmärrettävää. Siksi pienet kokonaisuudet on parempi kuvata tähän samaan malliinsa. [5]

Sijoitukset välilehdellä näytetään käyttäjän sopimukset. Jos sopimuksia on monta, näytetään listaus käyttäjän kaikista sopimuksista. Jos sopimuksia on vain yksi, ohjataan käyttäjä suoraan tälle sopimussivulle. Tässä työssä keskitytään Vaurastumisen palvelun sopimukseen. Sopimuksella on pääsivu, jossa on erinäisiä linkkejä, joista aukeaa lisä-

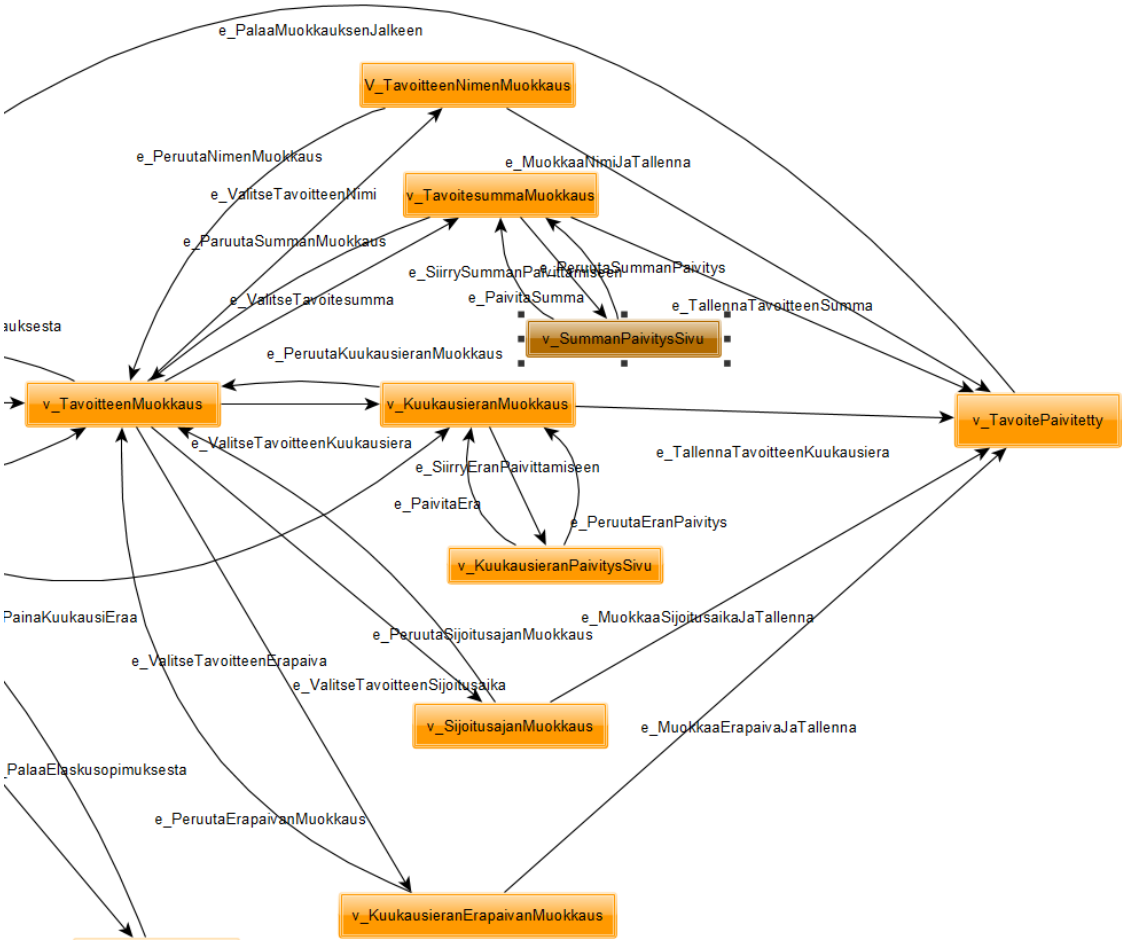
tietoja sopimuksesta. Aloitetaan jälleen mallinnus käymällä linkkejä läpi. Yksinkertaisimmat linkit avaavat vain uuden sivun, jossa on jotain tietoa sopimukseen liittyen. Näitä sivuja on muun muassa sopimuksen maksut -sivu, jossa on listattu käyttäjän sopimukseen maksamat maksut ja sijoituskohteet sivu, jossa listataan sopimukseen kuuluvat sijoituskohteet, eli mihin sopimuksen rahat on sijoitettu. Nämä linkit ja sivut on helppo mallintaa siirtymiksi ja tiloiksi. Sivuja on mallinnettu kuvassa 13.



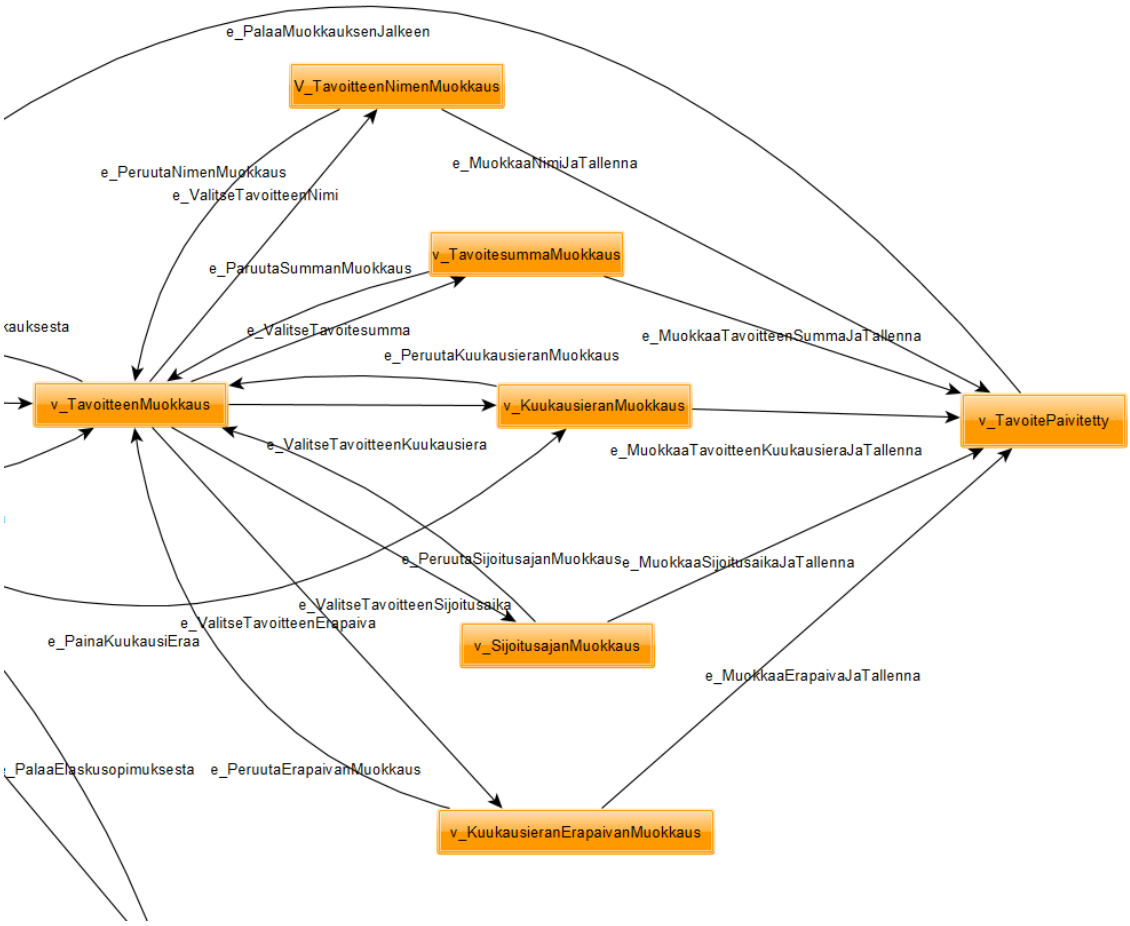
Kuva 13. *Vaurastumisen palvelun sopimuksen sivuja mallinnettuna yEdillä*

Vaurastumisen palvelun sopimuksen mielenkiitoisimmat ominaisuudet testauksen näkökulmasta ovat sopimuksen tavoitteen muokkaaminen ja sopimuksen maksujen maksaminen. Tavoitteen muokkaamisessa voi muokata tavoitteen nimeä, kuukausimaksua, eräpäivää, tavoiteaikaa tai kuukausimaksun eräpäivää. Tavoitteen muokkaaminen on monimutkainen kokonaisuus, jos sen mallintaisi tarkasti. Esimerkiksi tavoitesumman muokkaamisessa avautuu ensin sivu, jossa näytetään nykyinen summa ja miten se vaikuttaa tavoitteen saavuttamiseen. Summaa painaessa avautuu uusi sivu, jossa syötetään uusi muokattu summa. Lopuksi uuden muokatun summa voi tallettaa tai muokkauksen voi peruuttaa. Tässä tapauksessa siirtymiä ja tiloja tulee monta, joten on syytä yksinkertaistaa tilannetta koskemaan vain kiinnostavimpia testitapauksia. Mallinnetaan vain tilanteet, joissa tavoitetta muokataan ja muutos perutaan, sekä tavoitteen muokkaa-

minen ja muutoksen tallettaminen. Tätä tarkemmin ei yksittäistä muokkausta tarvitse kuvata. Toteutetaan samanlainen mallinnus kaikille tavoitteen muokkausoperaatioille. Kuvassa 14 on kuvattu tavoitteen muokkaaminen ennen yksinkertaistamista. Kuvassa 15 näkyy tavoitteen muokkaaminen yksinkertaistamisen jälkeen.



Kuva 14. *Vaurastumisen palvelun sopimuksen tavoitteen muokkaus kuvattuna tarkasti yEdissä*



Kuva 15. *Vaurastumisen palvelun sopimuksen tavoitteen muokkaus yksinkertaistamisen jälkeen yEdissä*

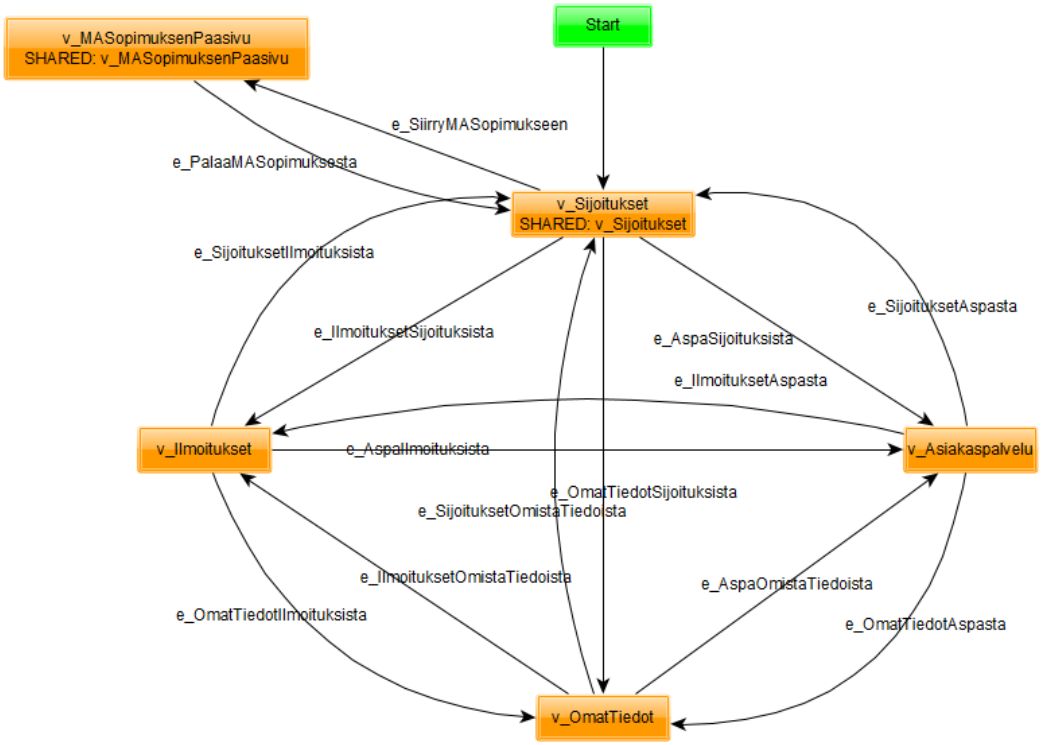
Toinen iso ominaisuus Vaurastumisen palvelun sopimuksessa on sopimuksen maksujen maksaminen. Maksuja on kahden tyyppisiä, kuukausimaksun maksaminen sekä kertasijoituksen tekeminen. Kuitenkin polut ovat suurin piirtein samanlaisia. Maksuoperaatio tapahtuu jälleen IAB:ssa, joten samat tapaukset tulee ottaa huomioon kuin aikaisemmin pankkitunnistautumisessa. Jälleen voidaan hieman yksinkertaistaa prosessia, sillä IAB:n sulkeminen on jälleen mahdollista missä vaiheessa tahansa maksua. Mielenkiintoisimmat tilanteet ovat onnistunut maksu, maksun peruuttaminen pankin päässä sekä selaimen sulkeminen kesken maksuprosessin. Kaksi jälkimmäisestä tilannetta ovat virhetapauksia ja niissä tulee näyttää virhesivu. Tilanteet on mallinnettu kuvassa 16.

Kuva 16. *Kuukausimaksun mallinnus yEdissä*

Vaurastumisen palvelun sopimus on sen verran iso kokonaisuus, että se on hyvä olla oma mallinsa. Lisäksi jatkokehitystä ajatellen on hyvä, että kukin sopimus on oma mallinsa. Käyttäjät, joilla on monta sopimusta, saadaan helpommin mallinnettua, kun voidaan lisätä testitapauksen luomisessa uusi malli mukaan testitapauksen luonnissa, sen sijaan että tulisi muokata erikseen samaa mallia tiettyjä käyttäjiä varten. Lisäksi kokonaisuuden kannalta mallin ylläpitäminen on helpompaa, kun sopimukset ovat erillisiä malleja. Tällöin on helpompi muokata vain tiettyä yksittäistä mallia, joka koskee tiettyjä käyttäjiä, kuin että jouduttaisiin muokkaamaan kaikkia käyttäjiä koskevaa mallia.

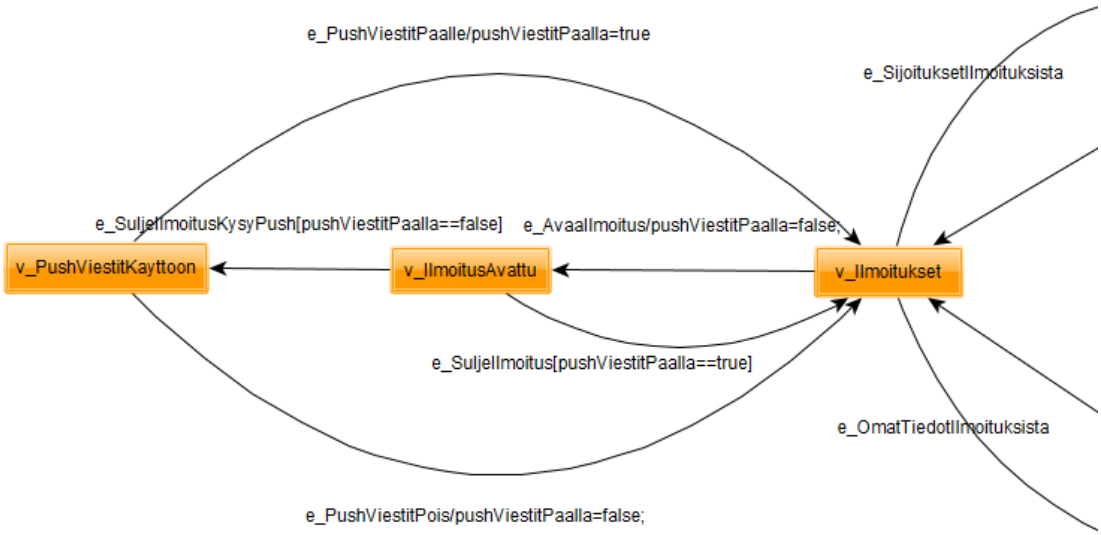
Sijoitussivu oli siis Vaurastumisen palvelun sopimus tässä tapauksessa. Koko malli on nähtävissä liitteessä B. Jäljellä on vielä kolme muuta alanavigaation painiketta mallinnettavaksi, jotka kuvataan samaan malliin. Huomionarvoista on myös itse alanavigaatiopalkin mallinnus. Periaatteessa joka sivulta voi tämän palkin avulla siirtyä mihin tahansa muuhun sivulle. Myös joiltain Vaurastumisen palvelun sopimuksen sivuilta voi siirtyä alanavigaation avulla muille sivuille. Mallin yksinkertaisuuden, ymmärrettävyyden ja testipolkujen hallitsemisen vuoksi kuvataan alanavigaatio vain niin, että jokaisen osa-

alueen pääsivulta voidaan siirtyä alanavigaation avulla toiselle sivulle. Kuvassa 17 on kuvattu alanavigaation mallinnus sekä siirtymineen Vaurastumisen palvelun sopimukseen.



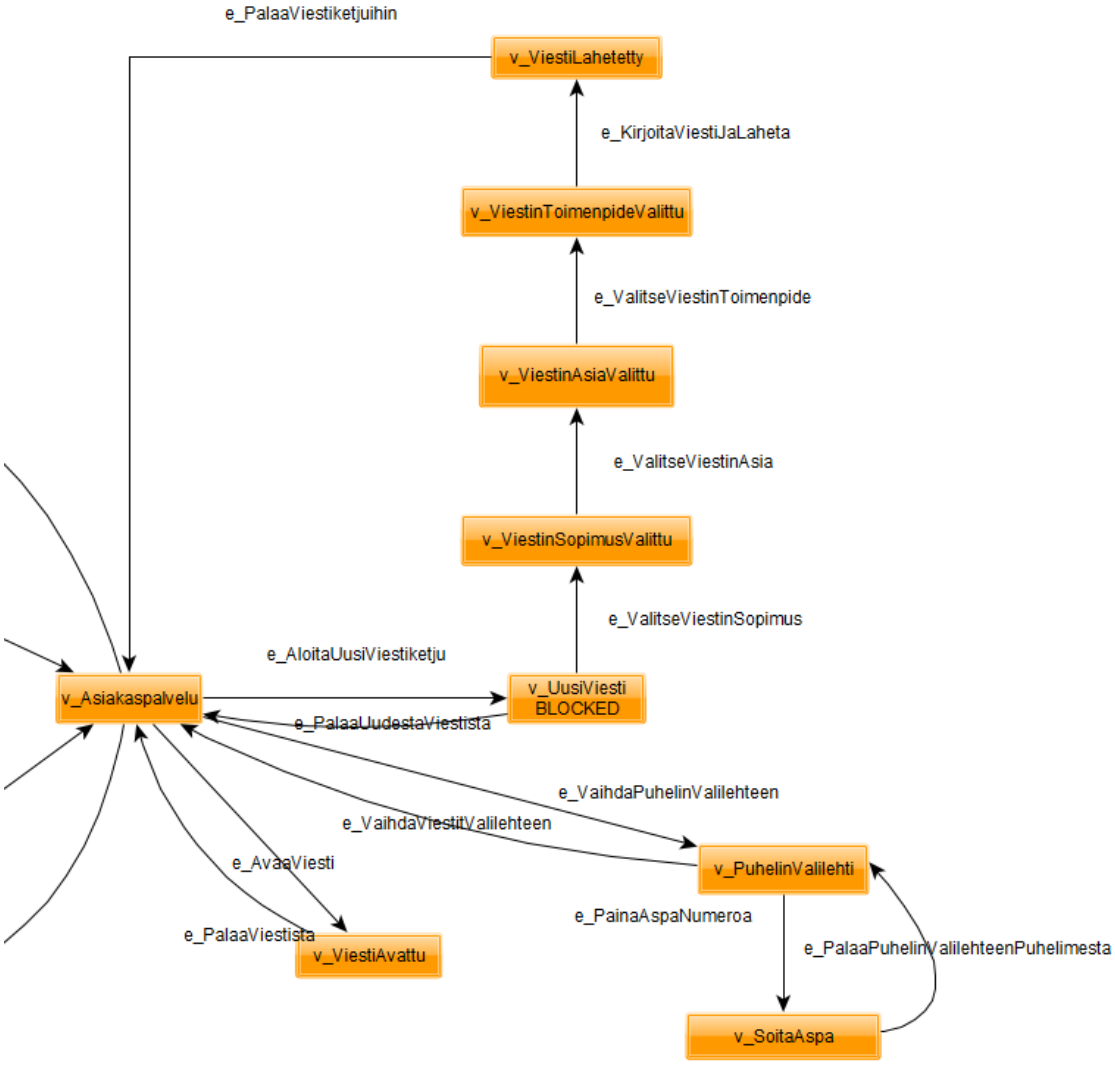
Kuva 17. *Alanavigaation malli ja vaurastumisen palvelun malliin siirtyminen yE-dissä*

Ensimmäinen kolmesta jäljellä olevista alanavigaation kohteista on ilmoitukset. Ilmoitussivulla näkyy käyttäjän saamat ilmoitukset. Ilmoituksia voi olla yksi tai useita, ja jokaisen ilmoituksen testaaminen erikseen vaatisi tietoa, mitä ilmoituksia käyttäjä on saanut. Tämän voisi kysellä tietokannoista, mutta tässä vaiheessa voidaan mallia yksinkertaistaa sen verran, että kaikki ilmoituksia ja niiden sisältöä ei tarvitse kuvata malliin. Riittää, että yksi ilmoitus aukeaa normaalisti, eli keskitytään vain toiminnallisuuden testaamiseen. Ilmoituksen sulkeutuessa kysytään käyttäjältä, haluaako hän vastaanottaa push-ilmoituksia. Tätä ei kysytä, jos push-ilmoitukset ovat jo päällä. Tämä tulee huomioida mallinnuksessa, sillä on asetettava muuttujien avuilla ehdot näihin tilanteisiin, jotta oikea polku käyttäjälle on kuljettavissa. Kuvassa 18 on kuvattu ilmoitusten toiminnallisuus malliin.



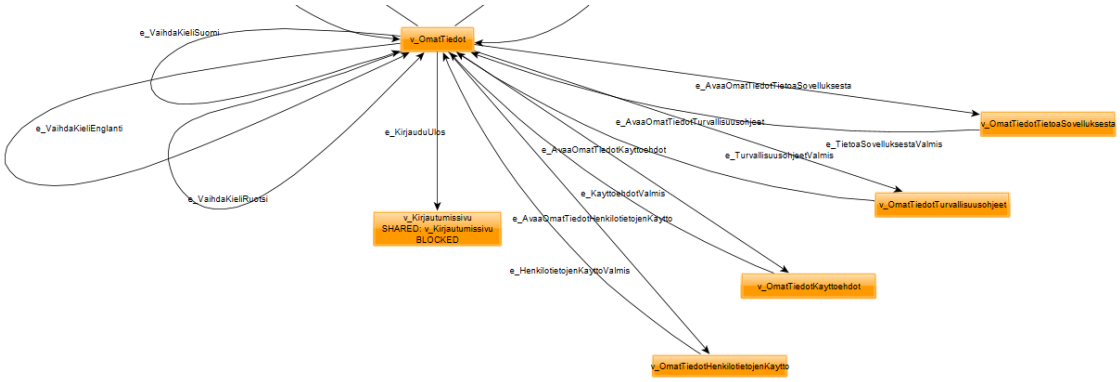
Kuva 18. *Yhden ilmoituksen toiminta mallinnettuna yEdissä*

Seuraava kohde alapalkissa on asiakaspalvelu. Tällä sivulla näytetään käyttäjän viestit, ja sivulla on kaksi välilehteä. Ensimmäisellä käyttäjä näkee omat lähetetyt ja vastaanotetut viestinsä asiakaspalvelun kanssa sekä voi lähettää uuden viestin asiakaspalveluun. Toisella välilehdellä on asiakaspalvelun tiedot ja sivulta voi soittaa asiakaspalveluun. Viestin lukeminen voidaan mallintaa kuten ilmoitus mallinnettiin aikaisemmin. Viestejä voi olla ilmoitusten tapaan useita, ja ilman tietoa muista järjestelmistä ei voida olla varmoja viestin sisällöstä, joten yksinkertaistetaan mallia jälleen kattamaan pelkästään viestin avaaminen ja sulkeminen. Uuden viestin lähettäminen on monivaiheinen prosessi, jossa valitaan viestin aihealue, mitä viesti koskee ja muita tarkennuksia. Koko prosessin ajan voi viestin lähetyksen keskeyttää palaamalla aiemmalle sivulle. Jälleen voidaan yksinkertaistaa mallia kuvaamalla vain yksi keskeytys, jotta prosessin keskeytys tulee testattua. Jokaisessa vaiheessa ei kannata erikseen kuvata keskeytystä, jottei testipolkujen määrä kasva liikaa [3]. Asiakaspalvelusivun tilat ja siirtymät on kuvattu kuvassa 19.



Kuva 19. *Viestien toiminta mallinnettuna yEdissä*

Viimeinen alanavigaation kohta on omat tiedot -sivu. Sivu sisältää käyttäjän omat tiedot, uloskirjautumisen, kielivalinnan sekä samoja lakitekstilinkkejä kuin infosivulla. Nämä siirtymät ja tilat on suoraviivaista mallintaa aiempaan tapaan. Kielivalinta on ainut hieman uusi asia. Se muuttaa käytännössä koko sovelluksen tilan eri kieliseksi, joten myöhemmässä vaiheessa se voi aiheuttaa ongelmia, kun tehdään Robot Framework-testejä. Jos jotain tekstiä käytetään varmistamaan, että ollaan oikeassa tilassa, tulee ottaa huomioon, että teksti voi vaihtua kielivalinnan mukaan. Tässä vaiheessa kuvataan vain jokainen kielivalinta malliin. Omat tiedot sivusta saatu kaaret ja solmut on kuvattu kuvaan 20.



Kuva 20. *Omat tiedot sivun malli piirrettynä yEdissä*

Sovellus on nyt mallinnettu kokonaisuudessaan yhdelle käyttäjälle. Mallien oikeellisuutta ja toiminnallisuutta voidaan testata Graphwalkerin avulla. Luomalla testipolkuja eri generaattoreilla, kuten ”random” ja ”a_star”, sekä eri pysähdysehdoilla kuten kaarikattavuus ja saavutetut solmut, nähdään, että Graphwalker saa luotua kokonaisia testipolkuja malleista. Jos malli ei ole oikeanlainen, jäädään polkua luodessa loputtomaan silmukkaan tai saadaan virheilmoitus. Tällöin korjataan mallia ilmoituksen mukaisesti. Mallia voi korjata myös myöhemmässä tilanteessa, mikäli törmätään virhetilanteeseen esimerkiksi testejä luodessa.

Malliin voidaan kuvata avainsanoilla estettyjä polkuja sekä kohtia. missä mallista voi hypätä toiseen malliin. Graphwalker osaa tulkita näitä tilanteita, jos solmuun kirjoittaa sanan ”BLOCKED” estetyille solmuille, sekä ”SHARED” solmuille, joista pääsee toiseen malliin. Tällöin tulee kirjoittaa myös sen solmun nimi mihin on mahdollista hypätä. Lisäksi tuetaan ehdollisia estoja, vahteja, joita voidaan muuttujien avulla käyttää arvioimaan tulisiko jonkin kaaren olla mahdollista suorittaa. Vahdit ja muuttujat ovat Javascript-koodia. Vahdit kuvataan hakasulkeisiin solmujen nimien perään yhtäsuuruusehdolla johonkin muuttujaan tarkasteltuna, ja itse muuttujat kuvataan kaarissa sen nimen perään kauttaviivan jälkeen Javascript-koodilla. [20]

4.5 Robottitestien toteuttaminen mallin pohjalta

Kun malli on todettu oikeanlaiseksi, voidaan alkaa toteuttamaan Robot Frameworkin avulla mallin siirtymiä ja tiloja. Toteuttamalla kaaret ja solmut Robot Framework syntaksiin avainsanoiksi, voidaan Graphwalkerin luomaa testipolkua ajaa robottitesteinä.

Alkuun siis luodaan Graphwalkerilla polku mallista, joka talletetaan tiedostoon. Tätä polkua voidaan sitten käyttää regressiotestinä, sillä se pysyy samana, kunnes tarvitsee luoda uusi polku mallin muuttuessa. Robottesti voidaan toteuttaa myös siten, että se luo aina uuden polun testiä ajettaessa. Tämä sopii hyvin, kun testataan vielä kehittämisen

alla olevaa sovellusta. Tällöin onkin tarpeen luoda monia eri testitapauksia, jotta testauksesta tulee mahdollisimman kattavaa. Usein myös tässä tilanteessa malli ja itse sovellus on vielä keskeneräinen, jolloin on tarpeen estää mallista tiettyjä polkuja, joten samaa polkua ei edes voi käyttää koko ajan.

Tehdään ensin silmukkarakenne, joka käy läpi testipolkua talletetusta tiedostosta lukien sieltä avainsanan kerrallaan ja suorittaen sen. Tämä onnistuu Robot Frameworkin FOR avainsanalla, joka on yksinkertainen silmukka. Silmukan ehto käy läpi jokaisen avainsanan tiedostosta. Silmukka tarvitsee myös muutamia alustavia avainsanoja.

Ensimmäinen avainsana ”Get File” hakee kuljettavan testipolkutiedoston annetusta paikasta. ”Split to lines” pilkkoo tiedoston osiin, jotta jokainen testiaskel saadaan erikseen. ”Get Length” avainsana laskee testiaskelten määrän, jotta voidaan kuvata testin suorittamisen aikana, kuinka monta testiaskelta on suoritettavana kokonaisuudessaan. ”Set Variable” asettaa arvon yksi indeksille. Indeksiä käytetään seuraamaan kuinka mones testiaskel milläkin hetkellä on suoritettavana. Seuraava avainsana on itse ”FOR”-silmukka, joka käy läpi jokaisen rivin aikaisemmin pilkotusta tiedostosta. Silmukan sisällä ensin tulostetaan kuinka mones testiaskel on suoritettavana, jonka jälkeen ajetaan sen hetkinen testiaskel testipolusta avainsanalla ”Run Keyword”. Lopuksi kasvatetaan indeksin arvoa yhdellä. Silmukan koko rakenne on kuvattu kuvaan 21.

```
41
42      ${path}=    Get File    ${PATHFILE_LOCATION}/${PATHFILE_NAME}
43      @{path_in_lines}=   Split to lines      ${path}
44      ${length}=  Get Length    ${path_in_lines}
45      ${index}=  Set Variable    1
46      :FOR    ${line}    IN    @{path_in_lines}
47      \       Log To Console  Running step ${index} out of ${length}: ${line}
48      \       Run Keyword    ${line}
49      \       ${index}=   Evaluate     ${index} + 1
50
51      [Teardown]    Run Keywords      Take screenshot     Close All Applications
52
53
```

Kuva 21. *Robottitestitiedoston silmukkarakenne, joka käy läpi tiedostoista löytyvät avainsanat ja ajaa ne*

Aloitetaan testien luominen ensimmäisestä luodusta mallista, eli kirjautumisprosessin mallista. Suurin osa kaarista ja solmuista on aika suoraviivaisia toteuttaa, Robot Frameworkin Appium-kirjaston valmiiden avainsanojen avulla voidaan klikata jotakin elementtiä. ML Rahat -sovelluksessa suurin osa kaarista onkin jonkin napin tai linkin painamista, joten voidaan käyttää avainsanaa ”Click Element”. Avainsana vaatii jokin paikantimen, jotta tiedetään mitä elementtiä painetaan. Elementin tunniste (id) on paras tähän, sillä se on aina yksilöllinen jokaiselle elementille. Aina tunnistetta ei ole saatavilla, jolloin

on pääteltävä jokin muu yksilöllinen tunniste elementille. Jos sovellusta on mahdollista tutkia Chrome verkkoselaimen avulla, osaa selain automaattisesti luoda yksilöllisen tunnisteen elementille Xpath-kielellä. Xpath on tarkoitettu kuvaamaan osia XML-kielestä, mutta sen avulla saadaan kuvattua myös HTML-tiedostojen osia, tässä tapauksessa elementtejä. [27]

Solmuissa varmistetaan, että on siirrytty oikeaan tilaan. Tällöin sivulta etsitään jokin tietty elementti ja varmistetaan, että se on nähtävillä sivulla, tai etsitään jokin tietty tekstinpätkä ja katsotaan, että teksti vastaa odotettua. Tekstin vertailu voi jossain tapauksissa olla huono testi, sillä kuten mainittiin aliluvussa 4.4, tekstin arvo voi muuttua sen mukaan, mikä kieli on valittuna. Voidaan kuitenkin toteuttaa tarkistus niin, että talletetaan käytetyn kielen arvo muuttujaan, jonka avulla saadaan verrattua oikean kielistä tekstiä odotusarvoksi. Kaksi solmuissa käytettyä avainsanaa ovat ”Wait Until Page Contains Element” ja ”Element Should Be Visible”. Nämä avainsanat vaativat myös lokaattorin jotta oikea elementti löytyy. Näiden avainsanojen avulla saadaan varmistettua, että haluttu elementti löytyy sivulta.

Kolmas usein käytetty avainsana on ”Input Text”, jolla saadaan syötettyä tekstiä tekstikenttään. ML Rahat -applikaatiossa tätä käytetään muun muassa käyttäjätunnuksen ja salasanan syöttämiseen. ”Input Text” -avainsana vaatii parametreiksi sekä lokaattorin että halutun syötettävän tekstin. Muutamia kaaria ja solmuja on kuvattu kuvassa 22.

```
56  e_PainaKirjaudu
57      [Documentation]    Paina kirjaudu nappia
58      Click element   id=button-login
59
60  v_Kirjautumissivu
61      # Ollaan kirjautumissivulla
62      Wait Until Page Contains Element    id=login
63      #Hide Keyboard
64      Element Should Be Visible   id=login
65      Element Should Be Visible   name=form_login
66
67  e_SyotaOikeatTunnukset
68      Input Text  id=user-id      266313
69      Input Text  id=password     2580
70
71  e_PainaInfoKuvaketta
72      Click element   id=button-info
73
74  v_InfoSivu
75      Wait Until Page Contains Element    id=info-page
76      Element Should Be Visible   id=info-page
77
78  e_PalaaInfostaAlkuun
79      Click element   id=button-back
80
```

Kuva 22. *Muutama esimerkki kaarista ja solmuista*

Koska ML Rahat -sovellus on hybridiapplikaatio, on kontekstia vaihdettava. Konteksti määrittelee mille asiayhteydelle Appium-käskyt menevät. Oletuksena käskyjä yritetään ajaa natiiviapplikaatioille, jolloin konteksti on ”native”. Hybridiapplikaatioille on valittava Webview.

Webview aiheuttaa hieman hankaluuksia, sillä vaikka Appium tukeekin myös hybridiapplikaatioita ja niiden testausta, on Appium-kirjasto suunniteltu pääosin natiiviapplikaatioille. Kaikki Appium-kirjaston tarjoamat avainsanat eivät ole siis tuettu hybridisovellusten testauksessa. Kontekstin vaihtoon on tarjolla valmis avainsana, ”Switch Context”. Se vaatii parametriksi halutun kontekstin nimen, joka tulee päätellä sovelluksesta. Apuna voidaan käyttää nykyisiä robottitestejä.

Suurin ongelma testien teossa aiheutuu sivun vierityksestä. Appium osaa yleensä automaattisesti selata sivua, kunnes haluttu elementti on näkyvillä, jotta sitä voi painaa. Mutta ML Rahat -applikaatio omaa alanavigaatiopalkin, joka pysyy koko ajan sivun alareunassa. Appium ei ymmärrä tätä ja yrittää selata sivua sen verran, että elementti tulee juuri näkyviin puhelimen ruudulle, vaikka todellisuudessa se jääkin navigaatiopalkin alle.

Tämä johtaa siihen, että Appiumin painallus tuleekin alanavigaatiopalkille, eikä halutulle elementille eli väärä elementti saa klikkauksen ja testiajo epäonnistuu. Tähän tulee siis kehittää oma ratkaisu, joka vierittää sivua sen verran, että elementti tulee näkyviin alanavigaatiopalkin alta.

Ratkaisu on käyttää suoraan Appium-kirjaston Python-versiota ja sen tarjoamia funktioita. Appium tukee omien kirjastojen tekoa, joiden funktiot näkyvät avainsanoina valmiiden avainsanojen tapaan. Kun ML Rahat -sovelluksen kontekstin vaihtaa Python-koodissa natiiviapplikaatioksi, saadaan ”swipe”-funktio toimimaan. Tämä funktio simuloi sitä, että käyttäjä laittaa sormen ruudulle ja pyyhkäisee ylöspäin. Tämä on ainoa funktio, mikä saatiin toimimaan lukuisista eri funktioista, joiden pitäisi selata sivua. Tämä vaikuttaa olevan yleinen ongelma Appiumin ja hybridiapplikaatioiden kanssa [28]. Valmis ”swipe”-funktiota hyödyntämä sivun selaus -funktio onkin toteutettu jo ML Rahojen nykyisiin automaattitesteihin, joten voidaan hyödyntää sitä. Kuvassa 23 on kuvattu Pythonilla toteutettu selausfunktio. Funktio pyyhkäisee alaspäin tai ylöspäin 1000 yksikköä. Yksiköiden todellinen yksikkö jää hieman epäselväksi, sillä vaikka sen pitäisi olla pyyhkäisyn loppu ja alkupisteen välimatka, eri lähteiden mukaan funktio ei välttämättä toimi dokumentoinnin mukaan [29].

```
290     def scrollaa(self, suunta, luku=1100):
291         """
292         XXXXXXXXXXXXXXXXXXXX
293         """
294         driver = self.__get_webdriver_instance()
295         bi = BuiltIn()
296         br = bi.get_library_instance("AppiumLibrary")
297         #print(br)
298         self.__change_context_to(0)
299         if suunta == "Alaspain":
300             br.swipe(100, int(luku), 100, 100)
301             #br.move_to(100, int(luku))
302         elif suunta == "Ylospain":
303             br.swipe(100, 100, 100, int(luku))
304         bi.sleep(2)
305         self.__change_context_to(1)
306         bi.sleep(2)
```

Kuva 23. *Selauksen Python toteutus ”swipe” funktion avulla*

Nyt kun sivua pystyy selaamaan, saadaan pääteltyä, onko elementti ruudulla näkyvissä vertailemalla niiden paikkaa ruudulla. Koska alanavigaatiopalkki pysyy ruudun alareunassa selatessa sivua, sen koordinaatit muuttuvat myös. Tarvitsee siis verrata, että halutun elementin alareunan y-koordinaatti on arvoltaan pienempi kuin alanavigaation yläreunan y-koordinaatti. Mikäli ei ole, on suoritettava pyyhkäisy uudelleen. Tästä saadaan

tehtyä silmukkarakenne, jonka tarkastusehto on kuvattu ehto. Silmukka suoritetaan maksimissaan kymmenen kertaa. Pyyhkäisyjä ei todennäköisesti tarvita edes niin monta kertaa, sillä jos elementti ei ole näkyvillä kymmenen kerran jälkeen, on kyseessä luultavasti virhetilanne. Toteutettu funktio on kuvattu kuvassa 24.

```
358  Scroll until element is not under navbar
359      [Arguments]    ${element_location}      ${element_size}
360
361      ${element_position}=    Set Variable    ${element_location}[y]
362      ${element_size_height}=  Set Variable   ${element_size}[height]
363
364      ${element_alareuna}=       Evaluate     ${element_position}+${element_size_height}
365
366      ${i}=  Set Variable    0
367      :FOR    ${i}    IN RANGE       10
368      \       ${navbar_location}=     Get Element Location    class=navigation
369      \       ${navbar_position}=     Set Variable    ${navbar_location}[y]
370      \
371      \       Exit For Loop If  ${element_alareuna} < ${navbar_position}
372      \       Scrollaa        Alaspain        1000
373      \       ${i}=   Evaluate     ${i} + 1
374
```

Kuva 24. *Vierittämisen robottitestien funktion toteutus*

Toinen hankala tilanne on IAB:n sulkeminen. Koska sen sulkeminen on sisäänrakennettu käyttöjärjestelmään, tulee IAB:ta käsitellä natiivissa kontekstissa. Sulkemisnappia saadaan klikattua sen tunnisteen perusteella, jolloin IAB sulkeutuu. On vaihdettava vielä takaisin Webview-kontekstiin, jotta seuraavien testien käskyt menevät oikealle kontekstille. Kun IAB avataan uudelleen sen jälkeen, kun se on jo kerran avattu seuraa häiriötilanne. Oletuksena kun IAB sulkeutuu, sen luoma uusi konteksti tuhotaan ja se poistuu kontekstien listauksesta. Todellisuudessa kuitenkin konteksti jää näkyviin listaukseen, sillä ML Rahat -sovelluksen käyttämässä teknologiassa itsessään on virhe, joka ei tuhoa IAB:ta täysin niiden sulkeutuessa [30]. Suljetun IAB:n kontekstien nimi muuttuu tyhjäksi, mutta kontekstit kuitenkin löytyvät niiden nimellä hakiessa. Tämä häiritsee testien suorittamista koska jos vaihdetaan väärään kontekstiin, saadaan virheilmoitus, koska kontekstiarvo on ”null”. Jos tällaiselle kontekstille yritetään kohdistaa Appium-komentoja, saadaan virheilmoitus ja testiajo epäonnistuu.

Tämä tilanne kierretään pienellä loogisella päättelyllä. Ensimmäiseksi avattu Webview-konteksti on listassa aina ensimmäisenä ja tätä ei tuhota missään vaiheessa koska se on itse sovelluksen käyttämä pääkonteksti. Tuorein avattu konteksti on aina listassa viimeisenä, sillä se on avattu viimeisenä. Voidaan siis päätellä, että konteksti mikä halutaan valita, on aina joko listan ensimmäinen, jos halutaan takaisin ML Rahat -sovelluksen pää-Webview’hun, tai viimeinen mikäli halutaan valita juuri avattu IAB Webview. Tämä

tarkistus voidaan poistaa, kun juurisyy käytettävässä teknologiassa korjataan. Kuvassa 24 näkyy tarkistusehto kontekstin vaihdossa.

```
529
530     def change_window_handle_to_title(self, title):
531         bi = BuiltIn()
532         br = bi.get_library_instance("AppiumLibrary")
533         bi.sleep(5)
534         driver = br._current_application()
535         handles = driver.window_handles
536
537         size = len(handles)
538         print ("Found " + str(size) + " webview windows: ")
539         for x in range(size):
540
541             if handles[x] is None:
542                 print ("Window handle is null, continuing to next one")
543                 continue
544
545             if x == 0 or x == size-1:
546                 driver.switch_to.window(handles[x])
547                 print ("Switched to window: " + driver.title)
548
549                 if driver.title == title:
550                     print ("Driver title '" + driver.title + "' matches parameter title '" + title + "' , breaking loop")
551                     break
552             else:
553                 print ("Looking for window that is either the first or the last one and x was: " + str(x) + ", and size was: " + str(size))
554                 continue
555
556         bi.sleep(1)
557
```

Kuva 24. *Kontekstin valinta*

Näiden ongelmien jälkeen, on suhteellisen suoraviivaista toteuttaa loput mallien kaaret ja solmut robot-testeiksi. Pääosin kaaret ovat jonkin napin tai linkin painamista ja solmut jonkin elementin löytämistä sivulla ja varmistamista että se näkyy käyttäjälle.

5. MALLIN KOKEILU JA TULOKSET

Kun malli ja sen testit ovat saatu toteutettua ja todennettua sen oikeellisuus, voidaan alkaa luomaan sen pohjilta testejä ja ajamaan kyseisiä testejä. Ensiksi on mietittävä, miten mallia halutaan ajaa ja miten sen testejä suoritetaan. Lisäksi tulee pohtia, minkälaisia testejä halutaan luoda. Mietittäviä asioita on muun muassa testiaskelten pituus ja kuinka kauan testien ajaminen kestää.

5.1 Mallin ajaminen

Mallin ja testien ajaminen on kaksivaiheista. Ensin tulee luoda halutuista malleista, joko yhdestä tai monesta, testipolku, joka talletetaan tiedostoon. Tämän jälkeen ajetaan robottitesti, joka lukee tämän polun tiedostosta ja suorittaa avainsanat tiedostossa kuvatussa järjestyksessä. Testipolkujen tallettaminen tiedostoon mahdollistaa niiden käytön myöhemmin, jos halutaan ajaa samoja polkuja uudelleen regressiomielessä.

Mallia voi ajaa myös suoraan robottitesteillä. Tällöin halutut mallit ja testien generoinnissa käytetyt optiot tulee antaa robottitestin tiedostossa muuttujina. Tämä vaihtoehto on tarkoitettu kehityksen aikaiseen testaukseen.

5.2 Testipolkujen luominen

Graphwalkeria ajetaan ajamalla sen jar-tiedostoa ja antamalla sille parametreja. Komennon syntaksi on antaa ensin jar-tiedoston polku, sitten käytetty menetelmä eli offline. Tämän jälkeen annetaan optio –m ja haluttu malli ja sitten jokin generaattori ja pysäytysehto, jolla mallista luodaan testipolku. Esimerkkikomento on kuvattu alla. Tämä komento luo polun, jossa mallista ”mallinnimi” luodaan satunnaisesti sen läpi kulkemalla testipolku, joka kattaa sata prosenttia mallin siirtymistä.

```
gw -m mallinnimi.graphml ”random(edge_coverage(100))”
```

Algoritmeja on monia kuten ”random”, joka satunnaisesti kulkee mallin läpi ja ”a_star” joka laskee lyhimmän reitin annettuun tilaan tai siirtymään. Pysäytysehtoja on muun muassa ”edge_coverage”, joka pysäyttää testipolun luomisen, kun annettu prosenttiluku siirtymiä on katettu, ”vertex_coverage” joka toimii vastaavasti tiloille ja ”reached_vertex”, joka pysäyttää polun luomisen, kun annettu tila on saavutettu. Ehtoja voi myös ketjuttaa yhteen. Alla on kuvattu komento, joka kulkee satunnaisesti mallin läpi, kunnes 55 prosenttia tiloista on katettu ja on saavutettu tila ”v_tila”.

```
gw offline -m mallinnimi.graphml ”random(vertex_coverage(100) && reached_vertex(v_tila))”
```

Testipolkuja voidaan luoda moneen malliin, tai vain yhteen. Pelkän kirjautumismallin testaus onnistuu luomalla pelkästään sille erilaisia polkuja, mutta muut mallit ovat riippuvaisia tästä mallista. Tällöin on ensin luotava kirjautumismallille polku, joka kulkee siihen tilaan, josta päästään hyppäämään toiseen malliin. Alla oleva komento luo testipolun kirjautumismallin läpi ja luo testipolun sovelluksen rungon mallille kattaen 55 prosenttia siirtymistä.

```
gw offline -m kirjautumismalli.graphml ”a_star(reached_vertex(v_Sijoitukset)) –m runkomalli.graphml “random(edge_coverage(55))”
```

Komennossa luodaan siis ensin lyhin polku v_sijoitukset tilaan, josta hypätään runkomalliin ja luodaan polku, joka käy läpi 55 prosenttia sen siirtymistä.

Tässä työssä käytetään lähinnä ”random” generaattoria ja ”edge_coverage” pysäytysehtoja. Nämä ovat tässä vaiheessa riittäviä kokeilemaan mallien ja testien toimintaa. Myöhemmässä vaiheessa voidaan testipolkujen suunnittelua miettiä tarkemmin, mutta kokeilemalla eri komentoja huomataan, että siirtymien kattavuuden perusteella luotavat testitapaukset kattavat suurimman osan tärkeimmistä testitapauksista. Kokeilemalla saadaan selville että 80 prosentin siirtymäkattavuus luo sopivan pituisia testipolkuja suoritettavaksi. Ne kattavat ison osan sovelluksen toiminnollisuuksista, eivätkä luo liian pitkiä polkuja. Testipoluista muodostuu näillä ehdoilla noin 200–300 askelta.

5.3 Robottitestien ajaminen

Robottitestejä ajetaan suoraan komentokehotteesta. Tällöin tulee olla siirtyneenä komentokehotteella samaan kansioon missä testitiedosto sijaitsee. Esimerkkikomento on kuvattu alla.

```
robot tiedoston_nimi.robot
```

Komento siis ajaa ”tiedoston_nimi” nimisen robottitestin. Ajon jälkeen muodostuu automaattisesti lokitiedosto, johon on kuvattu kaikki avainsanat, joita ajon aikana on kutsuttu. Mikäli avainsana on suoritettu onnistuneesti, se on väriltään vihreä. Mikäli avainsanan suorittaminen on epäonnistunut, se on kuvattu punaisena. Esimerkkiajoja on liitteissä D, E ja F.

Robottitestitiedoston rakenne on silmukka, joka käy läpi luotua testitiedostoa ja ajaa siinä kuvattuja avainsanoja. Jokaisen mallin avainsanat on toteutettu omiin tiedostoihinsa, ja ne voidaan tuoda päätiedostoon kirjastoina. Käytäessä tiedostosta saatua polkua läpi osaa robotti etsiä oikean avainsanan tuoduista kirjastoista.

Testejä ajetaan muutamia kertoja tarkkaillen samalla suoritusta. Jos törmätään virhetilanteeseen, tulee arvioida, onko virhe mallissa, robottitesteissä vai itse testattavassa sovelluksessa. Aloitetaan testaamalla kirjautumismallin pohjalta. Luodaan testipolku, joka kattaa 80 prosenttia siirtymistä, sillä se luo sopivan mittaisen polun, joka kattaa ison osan testitapauksista.

Ajossa törmätään yhteen virheeseen. Kun info sivulta siirrytään asiakaspalvelun numeroa painamalla puhelinsovellukseen, kopioituu numero eri ajoissa eri tavalla. Tämä on aika pieni häiriö, mutta voi estää pahimmassa tapauksessa asiakkaan soittamisen asiakaspalveluun. Kirjataan häiriö ylös ja ohitetaan tämä tapaus, jotta testiä päästää jatkamaan. Muihin virheisiin ei törmätä ja testit saadaan ajettua läpi onnistuneesti. Tämä testiajo sisältää infosivun linkkien avaamisen, väärällä tunnuksella kirjautumisen sekä tunnuksen luomisen.

Siirrytään seuraavaksi Vaurastumisen palvelun sopimuksen malliin. Tehdään samanlainen testiajo tämän mallin pohjilta. Nyt törmätään välillä virheilmoituksiin, sillä ympäristön hitaudesta johtuen ei sopimuksen etusivu ehdi aina latautua, ennen kuin testien aikakatkaisu lopettaa testin. Väliaikaisesti voidaan nostaa aikakatkaisua, mikä korjaa tilanteen. Myös toiseen virheeseen törmätään. Jos käyttäjä palaa pankin valinta sivulta maksun syöttöön, tyhjenee sopimuksen tilinumero ja viitenumero. Jos sivulta poistuu sopimuksen pääsivulle ja palaa takaisin, tilanne korjaantuu. Kirjataan myös tämä virhe ylös ja jatketaan testiä. Tämän jälkeen saadaan ajettua testiajo onnistuneesti. Ajo sisältää muun muassa tavoitteen muokkauksen osittain sekä sopimuksen yksittäisten sivujen avaamisen ja niistä palaamisen.

Testataan vielä runkomallin pohjilta luotua testipolkua samoilla ehdoilla luotuna. Törmätään samaa virheeseen kuin aikaisemmin, eli puhelinnumeron kopioitumiseen eri tavalla. Voidaan jatkaa testiä ja ohittaa tämä virhe, sillä se on jo kirjattu ylös. Muihin virheisiin ei törmätä ja saadaan ajo läpi onnistuneesti. Ajo sisältää push-viestin avaamisen sekä viestin lähetyksen ja avaamisen. Eri mallien ajojen perusteella voidaan olla vakuuttuneita, että mallit ja niiden testit ovat toimivia. Muutama virhe löytyi ja ne kirjattiin ylös.

5.4 Löydetyt virheet

Virheitä löytyi sekä mallista ja testeistä, että itse testattavasta sovelluksesta. Testiohjelman suurimmat viat ja ongelmat olivat kontekstien vaihdossa ja sivun selaamisessa. Kontekstin vaihto on periaatteessa sovelluksessa käytetyn teknologian virhe, joka tulee kiertää testeissä, jotta ne toimisivat. Selaaminen taas johtui Appiumin ja hybridiapplikaatioiden yhteensopimattomuudesta, joka pystyttiin kiertämään testien toteutusvaiheessa.

Itse testien kohteena olevasta sovelluksesta löytyi muutama vika. Kun käyttäjä siirtyy asiakaspalvelun numeroa klikkaamalla puhelinsovellukseen, kopioituu asiakaspalvelun numero puhelimeen eri tavalla eri ajoissa. Pahimmassa tapauksessa tämä estää käyttäjän yrityksen soittaa asiakaspalveluun, kun puhelinnumero on väärässä muodossa. Toinen löydetty vika löytyi Vaurastumisen palvelun sopimuksesta. Palatessa pankkien valinta sivulta maksun syöttö sivulle, häviää sivulta tili- ja viitenumero. Ne kuitenkin palasivat näkyviin, mikäli käyttäjä käy sopimuksen etusivulla ja valitsee maksun uudelleen.

Sovelluksesta löytyvät virheet olivat vakavuudeltaan pieniä, sillä ne ovat kierrettävissä eivätkä täten estä mitään toiminnollisuutta sovelluksesta. Kuitenkin alkuperäisen oletuksen vastaiseksi löydettiin muutama virhe. Tämä tarkoittaa, että mallipohjainen testaus löysi sovelluksesta virheitä, joita ei aikaisemmin ollut löytynyt.

5.5 Vertailu nykyisiin automaattitesteihin

Jotta saadaan jokin käsitys siitä kuinka kattavat testit mallipohjainen testaus voisi tuottaa, voidaan vertailla mallipohjaista testausta olemassa oleviin automaatiotesteihin. Nykyiset automaatiotestit ovat regressiotestejä, joten ne eivät oletuksena testaa jokaista toimintaa sovelluksesta. Vertailua ei siis kannata pitää totuutena vaan lähinnä suuntaa antavana. Oletuksena on, että koska malliin on kuvattu sovelluksen toiminnot kattavasti testauksen näkökulmasta, voi sen avulla saada aikaan kattavammat testit kuin nykyisillä automaatiotesteillä.

Vertailun tavoitteena on saada selville, kuinka paljon kattavammat testit mallipohjaisella testauksella voisi teoriassa saada aikaan. Suoritetaan vertailu kuvaamalla nykyiset automaatiotestit malleiksi toteutettujen mallien pohjilta. Tutkitaan siis nykyisiä testejä ja mietitään mitä ominaisuuksia ne eivät testaa mistäkin mallista, ja poistetaan nämä tilat ja siirtymät mallista. Lopuksi voidaan vertailla varsinaisten mallien ja nykyisten testien pohjilta luotujen mallien tilojen ja siirtymien määrää. Tulokset ovat teoreettisia ja suuntaa antavia, sillä varsinaisiin malleihin on kuvattu siirtymät molempiin suuntiin tilojen välissä, kun taas nykyisissä automaatiotesteissä ei välttämättä siirrytä saavutetusta tilasta enää takaisin edelliseen tilaan.

Kirjautumismallissa saatiin 21 tilaa ja 36 siirtymää, nykyisten testien perusteella luodussa mallissa tiloja oli 14 ja siirtymiä 22. Nykyisistä testeistä puuttui muun muassa infosivu kokonaisuudessaan sekä kirjautumisprosessin virhetilanteita. Sovelluksen rungon mallissa on 21 tilaa ja 45 siirtymää, vastaavasti nykyisten testien pohjilta tehdystä mallissa on 14 tilaa ja 26 siirtymää. Iso ero siirtymissä selittyy alanavigaation läpikäynnistä, joka on kuvattu malliin tarkemmin kuin nykyisissä testeissä, jotka käyvät alanavigaation

kohteet läpi järjestyksessä. Muita puuttuvia ominaisuuksia ovat push-viestien hyväksymisen kysyntä käyttäjältä. Vaurastumisen palvelun sopimuksen mallissa on 21 tilaa ja 44 siirtymää. Nykyisten automaatiotestien pohjilta tehtyyn malliin saatiin 16 tilaa ja 29 siirtymää. Nykyisistä testeistä puuttuvia ominaisuuksia ovat kaikkien sopimuksen tavoitteiden muokkaaminen sekä sijoituskohteet sivun avaaminen.

Kirjautumismallin osalta nykyiset automaatiotestit kattavat vain 67 prosenttia mallintamisen tunnistamista tiloista ja 61 prosenttia siirtymistä. Sovelluksen rungon mallin mukaan nykyiset testit kattavat 67 prosenttia tiloista ja 58 prosenttia siirtymistä. Vaurastumisen palvelun sopimuksen kannalta nykyiset testit kattavat 72 prosenttia tiloista ja 66 prosenttia siirtymistä. Nämä luvut eivät ole kovin tarkkoja vaan ne ovat suuntaa antavia. Ne kuitenkin tukevat oletusta, että mallipohjaisen testauksen avulla saadaan kattavampia testitapauksia luotua. Tulokset ovat kuvattuina taulukoihin 1, 2 ja 3.

	Tilat	Siirtymät
Nykyiset automaatiotestit	14	22
Kirjautumismalli	21	36
Nykyisten testien kattavuus suhteutettuna malliin	67 %	61 %

Taulukko 1. Kirjautumismallin ja nykyisten automaatiotestin kattavuuden vertailu

	Tilat	Siirtymät
Nykyiset automaatiotestit	14	26
Sovelluksen rungon malli	21	45
Nykyisten testien kattavuus suhteutettuna malliin	67 %	58 %

Taulukko 2. Sovelluksen rungon mallin ja nykyisten automaatiotestin kattavuuden vertailu

	Tilat	Siirtymät
Nykyiset automaatiotestit	16	29
Vaurastumisen palvelun sopimuksen malli	21	44
Nykyisten testien kattavuus suhteutettuna malliin	72 %	66 %

Taulukko 3. Vaurastumisen palvelun sopimuksen mallin ja nykyisten automaatiotestin kattavuuden vertailu

5.6 Työmäärän vertailu

Työmäärän vertailussa arvioidaan mallipohjaisen testauksen työmäärää ja suhteutetaan se nykyiseen käytössä olevaan SAFe-kehitysmenetelmään. Menetelmässä kehitys tapahtuu sprinteissä, joissa ensimmäisessä sprintissä testauksen kannalta valmistaudutaan kehitysaikaiseen testaukseen luomalla testitapauksia. Ensimmäinen sprintti kestää noin kaksi viikkoa. Ideaalitilanteessa mallin ja osan testeistä tulisi saada tehtyä tämän sprintin aikana.

Tässä työssä mallin luomiseen meni noin viikko, josta iso osa ajasta meni sovellukseen tutustumisen parissa. Sovellus oli jo ennestään tuttu, mikä pienentää tutustumisaikaa. Kehitysympäristön pystyttämiseen meni noin kaksi tai kolme päivää. Tämän pitäisi kuitenkin olla kertaluontoinen asia, eikä tarvitse jatkossa huomioida työmäärään, jos mallipohjaista testausta päätetään jatkaa.

Testien luomiseen meni myös noin viikko aikaa. Suurimmat ongelmat veivät eniten aikaa, suurin osa testeistä oli suoraviivaisia toteuttaa. Viikossa saadaan siis ainakin perustapaukset testeistä toteutettua. Kehityksen valmistuessa myös testejä voidaan tehdä lisää.

Manuaalitestaukseen verrattuna vie mallipohjainen testaus enemmän aikaa alkuvaiheessa. Manuaalitestauksessa saadaan testattua toiminnallisuuksia sitä mukaa, kun ne valmistuvat. Mallipohjaisessa testauksessa puolestaan tulee toteuttaa malli ja testit ensin, ennen kuin testaus voidaan aloittaa. Mallipohjaisessa testauksessa työmäärä on siis alussa mahdollisesti isompi.

Tulosten perusteella mallipohjainen testaus pystyttäisiin sopeuttamaan nykyisin käytössä olevaan SAFe-menetelmää ainakin ajallisesti. Ensimmäinen sprintti tulisi siis käyttää kokonaisuudessaan mallin ja perustapausten testien luomiseen. Myöhemmissä sprinteissä mallia tulee päivittää ja testejä tehdä enemmän.

Samalla kun malli ja testit on toteutettu, saataisiin regressiotestit toteutettua. Luomalla mallin pohjalta sopivat testitapaukset, voidaan ne siirtää suoraan kehityksen jälkeen regressiotesteiksi, jolloin säästetään nyt regressiotestien tekemiseen menevä aika.

6. TULOSTEN ANALYSOINTI JA JATKOTOIMENPITEET

Jotta saaduista tuloksista on hyötyä, tulee niitä analysoida. Tulosten perusteella tulee miettiä, mitä hyötyä mallipohjaisesta testauksesta oli. Lisäksi tulee pohtia, miten tulosten pohjalta voidaan jatkossa kehittää malleja ja mallipohjaista testausta.

6.1 Tulosten analysointi

Tuloksista nähdään, että mallipohjaisuudesta on hyötyä. Se löysi ML Rahat -sovelluksesta virheitä, joita ei aikaisemmin ollut löydetty. Virheitä löydettiin kolmea eri tyyppiä eri osa-alueilta: ML Rahat -sovelluksen dokumentoinnista, itse ML Rahat -sovelluksesta, sekä itse mallista ja sen testeistä.

Mallipohjainen testaus auttoi usealla eri osa-alueella sovelluksen kehityksessä. Suurin hyöty saatiin mallin tekemisessä, jossa perehdyttiin sovelluksen toimintaan syvällisesti samalla kun sitä mallinnettiin. Itse mallinnusprosessi oli siis jopa hyödyllisempi kuin varsinaiset testiajot, joita ajettiin mallin pohjilta. Tämä toki riippuu testauksen kohteena olevasta sovelluksesta, sillä tässä työssä se oli jo toimiva ja asiakkaiden käytössä oleva sovellus eikä tavoitteena välttämättä ollut löytää kovin monta virhettä sovelluksen toiminnassa.

Mallipohjaisen testauksen tuloksena ovat mallit, joita voi käyttää osana sovelluksen dokumentaatiota, joka on tällä hetkellä hieman puutteellista. Mallin pohjalta voidaan sanoa, miten useimmissa tilanteissa sovelluksen tulisi toimia. Nykyisistä testeistä sitä ei voi päätellä, sillä ne eivät kata koko sovellusta, vain jotkin tietyt ydinalueet, sillä automaatiotestaus on nykyisin lähinnä regressiotestausta ja manuaalisesta testauksesta ei jää tarkkoja testitapausten kuvauksia. Mallista olisi siis erityistä hyötyä sovelluksen nykyisten määrittelyiden ja dokumentoinnin tukena.

Mallipohjaisessa testauksessa menee kauemman aikaa kuin manuaalitestauksessa, mutta lopputuloksena ovat kattavammat testit. Iso osa käytetystä ajasta menee ympäristön pystyttämiseen sekä mallin luomiseen. Ympäristön pystyttäminen pitäisi olla kerran suoritettava toimenpide, joka ei jatkossa vie aikaa varsinaiselta testaukselta. Testien toteuttaminen mallin pohjalta ei vie niin paljoa aikaa ja on työn perusteella aika suoraviivaista. Usein uuden kehityksen alkuvaiheessa on testaajilla aikaa luoda testitapauksia sovellukselle. Tämä aika voitaisiin käyttää myös mallien luomiseen, jos mallipohjaista testausta halutaan sovittaa mukaan nykyiseen prosessiin.

Mallipohjainen testaus vie manuaalitestaukselta aikaan. Molempia ei välttämättä pystytä tekemään samaan aikaa. Jos yhden testaajan pitäisi tehdä sekä mallit, toteuttaa mallin testit sekä vielä testata sovelluksen toimintaa manuaalisesti, ei tähän kaikkeen tule riittämään aika nykyisen mukaisessa kehitysprosessissa. Manuaalitestauksen merkitystä ja osuutta testauksessa voisi pienentää mallipohjaisen testauksen määrän kasvaessa. Manuaalitestausta pitää kuitenkin myös tehdä edelleen mallipohjaisen testauksen rinnalla, sillä on joitain osa-alueita, joita vain ihminen ymmärtää, kuten esimerkiksi sovelluksen ulkoasu. Sovelluksen malleja voi kuitenkin käyttää myös manuaalitestauksessa apuna määrittelyiden ja dokumentaation muodossa. Testaaja voi käyttää malleja manuaalitestitapausten luomiseen.

6.2 Jatkotoimenpiteet

Tulosten perusteella mallipohjainen testaus on kannattavaa, ja sen käyttöä tulisi jatkaa. Kun nyt toteutettiin mallipohjainen testaus olemassa olevaan järjestelmään, voisi kokeilla myös testata mallipohjaista testausta uuden kehityksen ohessa. Uusi toiminnallisuus testattaisiin siis sekä mallipohjaisesti että manuaalisesti. Kohteena tulisi olla ML Rahat -sovellus, sillä siihen on toteutettu jo malli. Mikäli kohteena olisi eri sovellus, olisi se mallinnettava alusta asti, mikä vie enemmän aikaa.

Myös ML Rahat -sovelluksen mallintamista tulisi jatkaa. Kun on nähty mallinnuksen hyödyt yhden sopimuksen testauksessa, voisi malleja laajentaa kattamaan myös muut palvelussa näkyvät sopimukset. Mallit ja testit tulisi myös muuttaa siten, että ne ottavat huomioon paremmin eri käyttäjät ja käyttäjien eri sopimukset. Esimerkiksi käyttäjästä voisi hakea tietoa muista järjestelmistä testiajon alussa, jolloin voisi tarkemmin testata käyttäjää koskevia tietoja. Näitä olisi esimerkiksi asiakaspalvelun viestit ja saapuneet ilmoitukset, jotka nyt ovat kuvattuna malliin pelkästään toiminnallisuuden osalta.

Mallipohjaisen testauksen voisi ottaa osaksi ML Rahat -sovelluksen regressiotestausta. Voitaisiin luoda joitain tiettyjä kattavia testipolkuja, jotka käyvät läpi ison osan sovelluksesta. Varmistetaan, että näiden testipolkujen ajot menevät läpi ja siirretään testipolut regressiotesteiksi. Lisäksi voitaisiin luoda jokin uusi testipolku automaattisesti esimerkiksi kerran tai kaksi päivässä osana automaattitestejä. Tällöin tulisi katsottua hieman eri polkuja kuin mitä yleensä tulee testattua. Tällöin kyse ei ole kuitenkaan enää regressiotestauksesta.

Kaikkiin sovelluksiin ei mallipohjainen testaus kuitenkaan sovi. Sovelluksen pitää olla riittävän yksinkertainen, jotta se on helppo mallintaa, muttei kuitenkaan liian yksinkertai-

nen. Tällöin mallipohjaisesta testauksesta ei ole etua, kun samat testitapaukset ovat nopeampi luoda ja testata manuaalisesti, ja aikaa kuluu mallin ja mallin testien luomiseen. Jos sovellus on taas liian monimutkainen, paisuvat siitä saatavat mallit erittäin monimutkaisiksi ja siitä saatavat testitapaukset kasvavat hallitsemattoman suuriksi. Tämä saattaa käydä myös muissa testausmenetelmissä, ja mallipohjaisen testauksen etuna onkin kuvalliset esitykset, jotka voi olla helpommin ymmärrettävissä kuin pitkät testitapaukset.

Jos mallipohjaista testausta halutaan soveltaa muihin sovelluksiin, tulisi ensin tutkia, mitkä niistä olisivat sopivia kandidaatteja mallipohjaiselle testaukselle. Sovelluksista voi myös mallintaa vain osan, jos on jokin tietty selkeä osa, missä mallipohjaisesta testauksesta olisi apua. Esimerkiksi sen sijaan, että mallinnettaisiin jokin verkkosivu kokonaisuudessaan, voidaan siitä mallintaa vain jokin tietty alasivu, jota halutaan testata tarkemmin.

7. YHTEENVETO

Tässä työssä toteutettiin mallipohjainen testaus esimerkkisovellukselle. Mallipohjaisen testauksen hyötyjä ja haittoja tutkittiin. Työn tavoitteena oli luoda sovelluksen malli ja saada sitä ajettua. Tämä saatiin toteutettua ja tulosten perusteella mallipohjainen testaus on kannattava testausmenetelmä jatkoa ajatellen. Vaikka mallipohjainen testaus on hieman työläämpää kuin nykyinen käytetty automaatiotestausmenetelmä tai manuaalitestaus, toisi mallipohjainen testaus hyötyä kattavampien testien ja dokumentaation muodossa.

Sovelluksen mallit ja testit toteutettiin avoimen lähdekoodin tai muuten ilmaiseksi saatavilla olevilla työkaluilla. Malli piirrettiin yEd-ohjelmalla, testit luotiin Grahpwalker-ohjelmalla mallien pohjilta ja itse automaatiotestit toteutettiin Robot Frameworkillä. Appiumia käytettiin apuna mobiilisovellusympäristön testauksessa.

Suurin osa työstä kului mallien ja testien toteuttamiseen. Mallinnus prosessina voi olla aikaa vievä, mutta se auttaa ymmärtämään sovelluksen toimintaa. Testien toteutus on taas nopeampaa työtä, mutta myös testien kirjoittamisen aikana voi tulla vastaan ongelmia tai löytää häiriöitä sovelluksen toiminnasta.

Työn tarkoitus oli tutkia, onko mallipohjainen testaus soveltuva mobiilisovelluksien testaukseen. Koska mallipohjaista testausta saatiin harjoitettua, voidaan tavoitteen sanoa olleen saavutettu. Tulosten mukaan mallipohjainen testauksen avulla saataisiin sovellukselle luotua kattavammat testit, joten mallipohjaisen testauksen hyödyt tuli todistettua ja sen käyttöä suositellaan jatkettavaksi

LÄHTEET

[1] P. C. Jorgensen, The Craft of Model-Based Testing, Auerbach Publications, 2017. Saatavissa: https://ebookcentral.proquest.com/lib/tampere/detail.action?docID=4856178

[2] I. Schieferdecker, Model-Based Testing, IEEE Software, 29(1), 2012, pp. 14-18. Saatavissa: https://search-proquest-com.libproxy.tuni.fi/docview/912094932?pq-origsite=summon. Viitattu: 6.4.2019

[3] A. Kramer, B. Legeard, Model-Based Testing Essentials: Guide to the ISTQB® Certified Model-Based Tester Foundation Level, Wiley-Blackwell, 2016. Saatavissa: https://www-books24x7-com.libproxy.tuni.fi/marc.asp?bookid=117500

[4] L. Apfelbaum, J. Doyle. Model Based Testing, Software Quality Week Conference, May 1997. Saatavissa: http://www.testoptimal.com/ref/Model%20Based%20Testing.pdf. Viitattu: 7.4.2019

[5] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, 2006. Saatavissa: https://www.sciencedirect.com/book/9780123725011/practical-model-based-testing

[6] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, T. Stauner, One evaluation of model-based testing and its automation, ACM, May 15, 2005, pp. 392-401. Saatavissa: https://dl-acm-org.libproxy.tuni.fi/citation.cfm?id=1062529. Viitattu: 6.4.2019

[7] G. de Cleva Farto, A. T. Endo, Evaluating the Model-Based Testing Approach in the Context of Mobile Applications. Electronic Notes in Theoretical Computer Science, 314, June 2015, pp. 3-21. Saatavissa: https://www-sciencedirect-com.libproxy.tuni.fi/science/article/pii/S1571066115000250. Viitattu: 12.4.2019

[8] V. Gudmundsson, M. Lindvall, L. Aceto, J. Bergthorsson, D. Ganesan, Model-based Testing of Mobile Systems – An Empirical Study on QuizUp Android App, Electronic Proceedings in Theoretical Computer Science, 208, 2016, pp. 16-30. Saatavissa: https://doaj.org/article/fe07a22672864ab389ed2aae9748eefd. Viitattu: 11.4.2019

[9] ML Rahat -mobiilipalvelu. Saatavissa: https://www.mandatumlife.fi/asiakaspalvelu/vakuutus-ja-sopimusasiat/mobiilipalvelu. Viitattu: 1.5.2019

[10] ML Rahat App Storessa. Saatavissa: https://itunes.apple.com/fi/app/ml-rahat/id933839623?l=fi&mt=8. Viitattu: 1.5.2019

[11] J. Cowart, What is a Hybrid Mobile App, Telerik, June 2012. Saatavissa: https://www.telerik.com/blogs/what-is-a-hybrid-mobile-app-. Viitattu: 2.5.2019

[12] Apache Cordova. Saatavissa: https://cordova.apache.org/. Viitattu: 2.5.2019

[13] A. Ziflaj, Native vs Hybrid App Development, Sitepoint, August 2014. Saatavissa: https://www.sitepoint.com/native-vs-hybrid-app-development/. Viitattu: 2.5.2019

[14] Mandatum Lifen Vaurastumisen palvelu. Saatavissa: https://www.mandatum-life.fi/vaurastu. Viitattu: 1.5.2019

[15] I. Blair, What is a Push Notification? And Why It Matters, Buildfire. Saatavissa: https://buildfire.com/what-is-a-push-notification/. Viitattu: 2.5.2019

[16] SAFe Scaled Agile Method. Saatavissa: https://www.scaledagileframe-work.com/. Viitattu: 4.5.2019

[17] SAFe Scaled Agile Method, Product Owner. Saatavissa: https://www.scaledagi-leframework.com/product-owner/. Viitattu: 4.5.2019

[18] SAFe Scaled Agile Method, Epic Owner. Saatavissa: https://www.scaledagilef-ramework.com/epic-owner/. Viitattu: 4.5.2019

[19] yEd Graph Editor, yWorks. Saatavissa: https://www.yworks.com/prod-ucts/yed/applicationfeatures. Viitattu: 5.5.2019

[20] Graphwalker. Saatavissa: https://graphwalker.github.io/. Viitattu: 5.5.2019

[21] Graphwalker-project commits, Github. Saatavissa: https://github.com/GraphWalker/graphwalker-project/commits/master. Viitattu: 5.5.2019

[22] fMBT. Saatavissa: https://01.org/fmbt. Viitattu: 1.4.2019

[23] RoboMachine. Saatavissa: https://github.com/mkorpela/RoboMachine. Viitattu: 1.4.2019

[24] Robot Framework. Saatavissa: https://robotframework.org/. Viitattu: 1.4.2019

[25] Appium. Saatavissa: http://appium.io/. Viitattu: 1.4.2019

[26] Selenium. Saatavissa: https://www.seleniumhq.org/. Viitattu: 1.4.2019

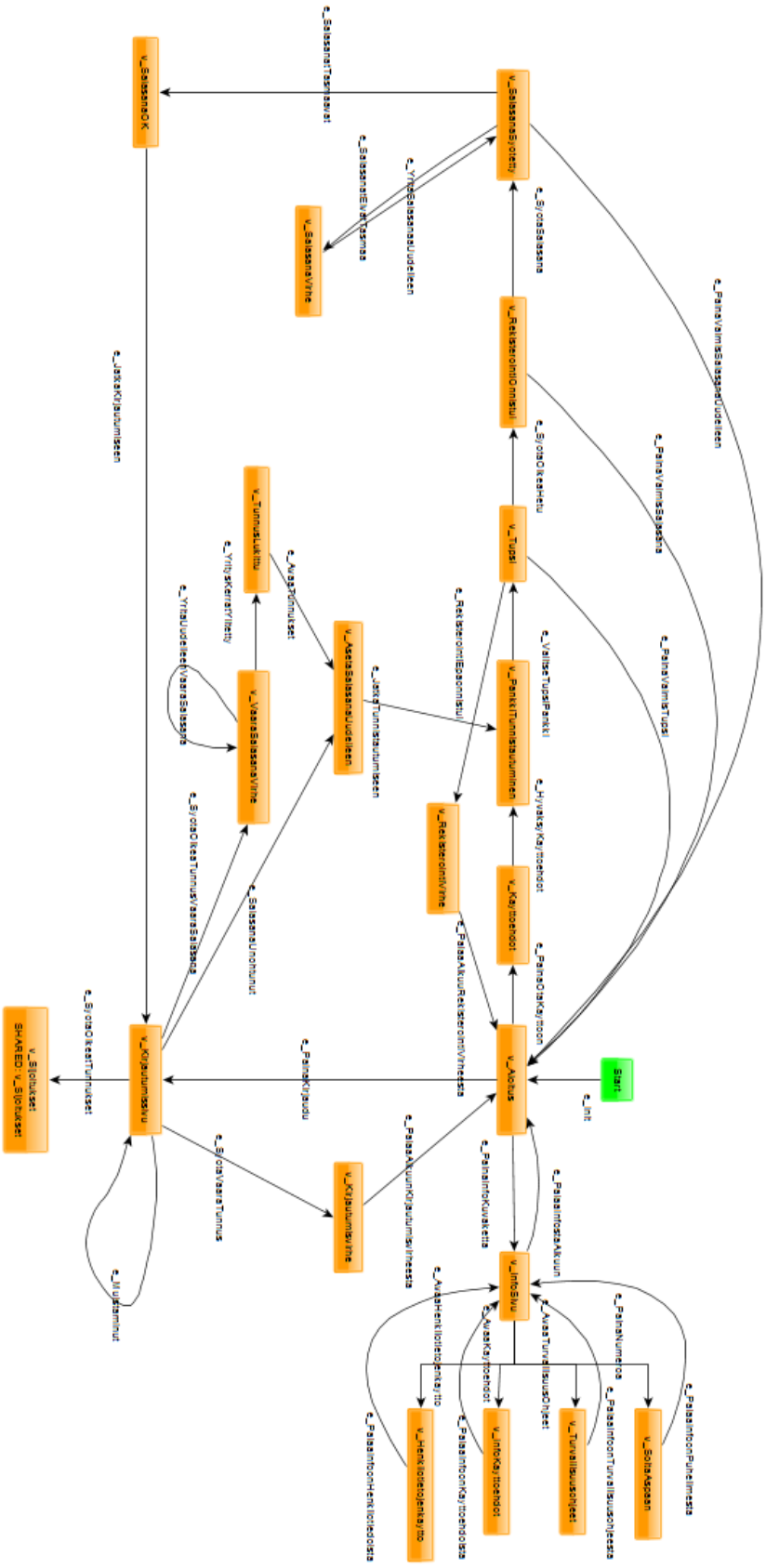
[27] XPath Tutorial, W3schools. Saatavissa https://www.w3schools.com/xml/xpath_intro.asp. Viitattu: 11.5.2019

[28] Robotframework - Tap, Swipe and Other “Mobile” Methods with Hybrid App on Angularjs, Stackoverflow. Saatavissa: https://stackoverflow.com/questi-ons/32118487/robotframework-tap-swipe-and-other-mobile-methods-with-hyb-rid-app-on-angula. Viitattu: 11.5.2019

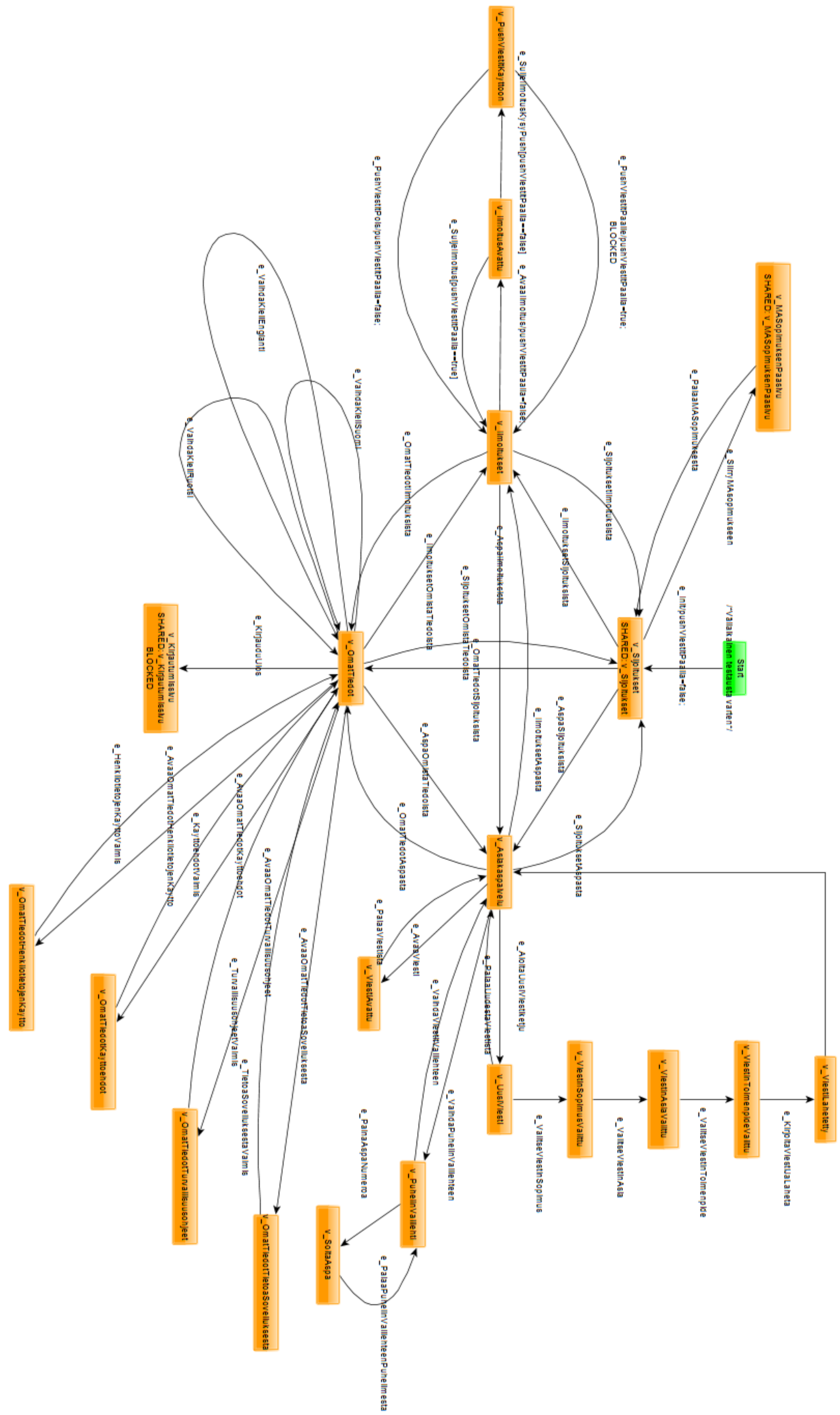
[29] Swipe Method Not Supporting in Android Appium. Stackoverflow. Saatavissa: https://stackoverflow.com/questions/29935366/swipe-method-not-supporting-in-android-appium. Viitattu: 12.5.2019

[30] InAppBrowser will not destroy WebView after being closed (by invoking ref.close()) causing memory leaks, Github. Saatavissa: https://git-hub.com/apache/cordova-plugin-inappbrowser/issues/290. Viitattu: 12.5.2019

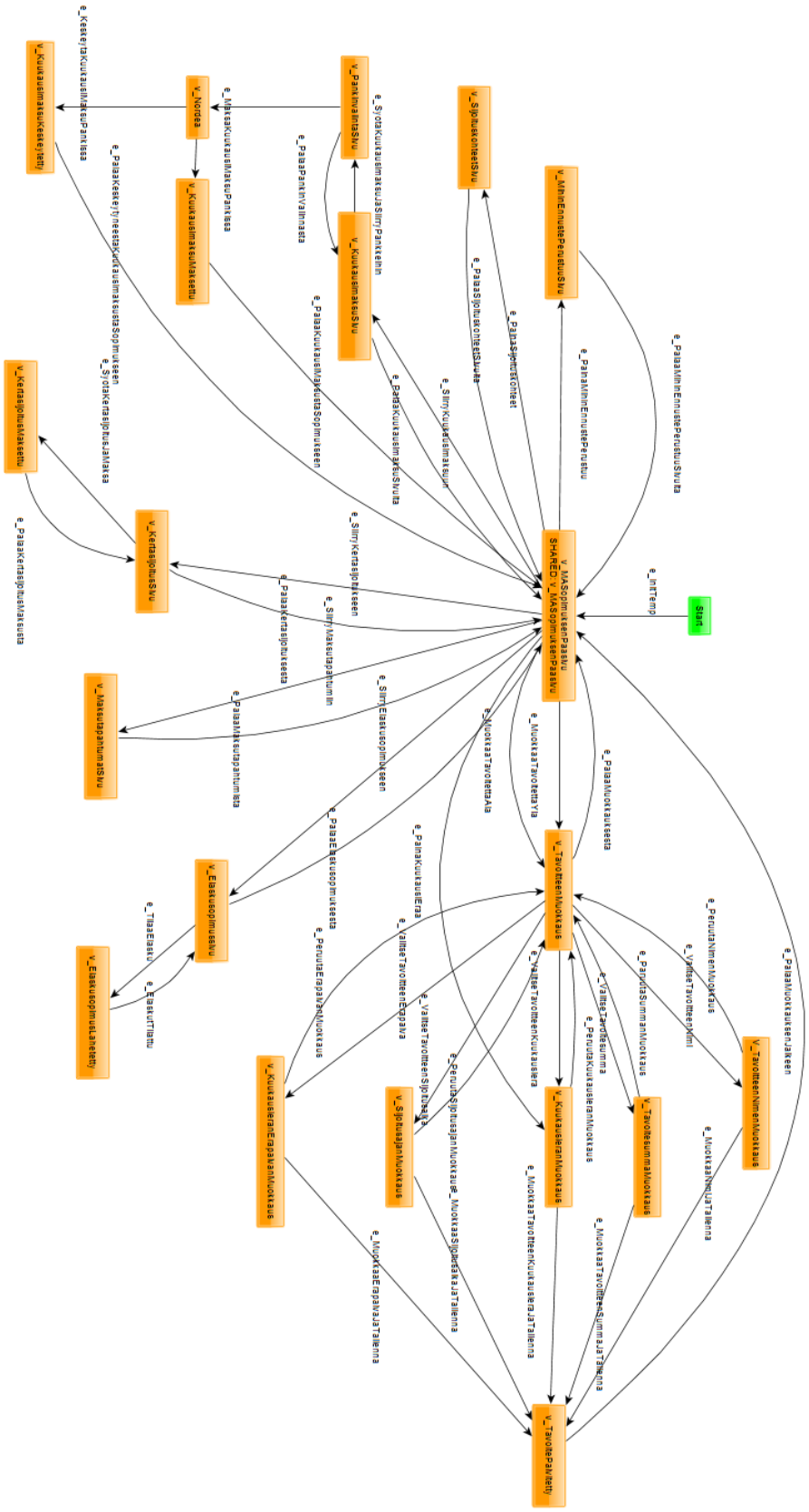
LIITE A: KIRJAUTUMISPROSESSIN MALLI



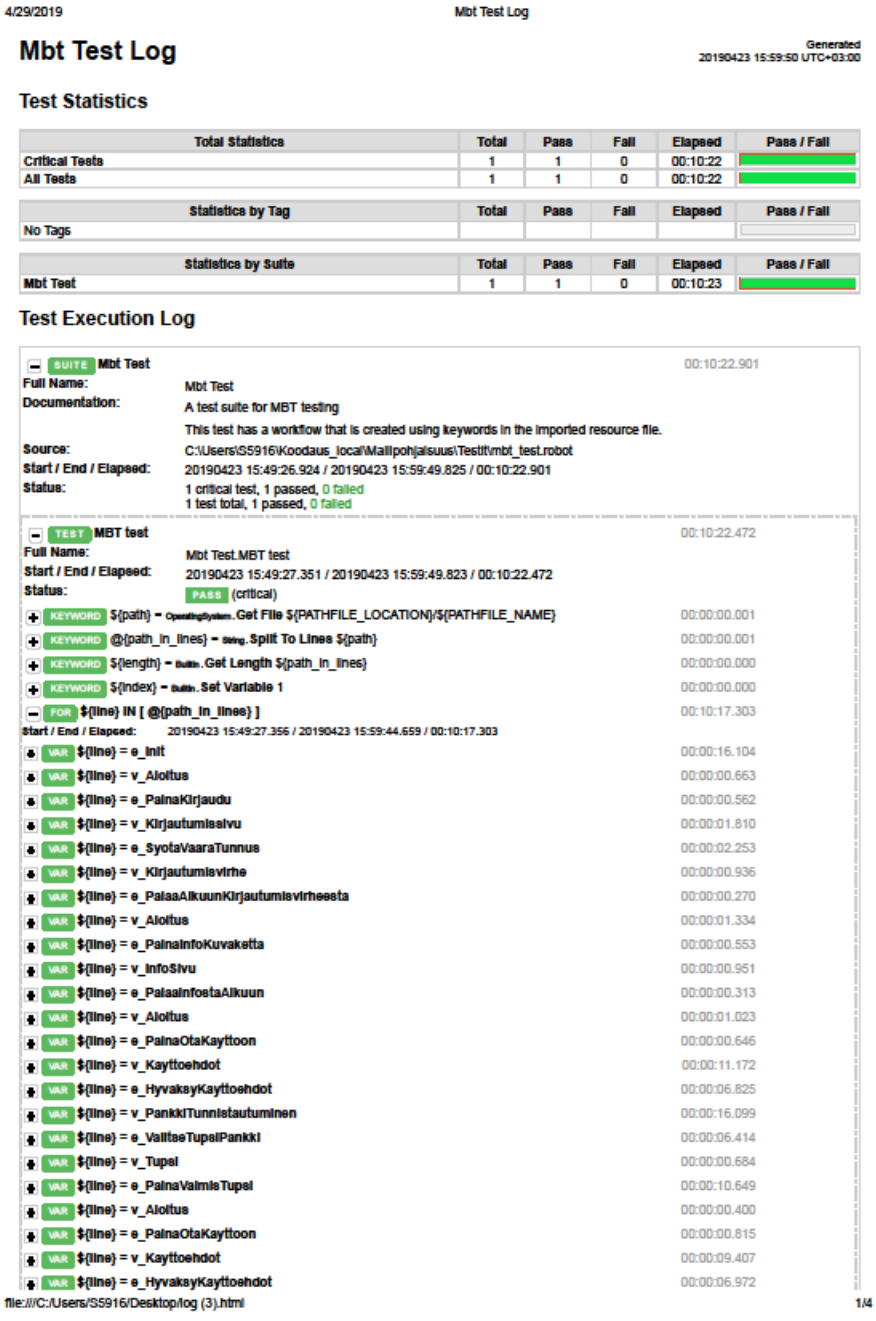
LIITE B: SOVELLUKSEN RUNGON MALLI



LIITE C: VARAUSTUMISEN PALVELU - SOPIMUKSEN MALLI



LIITE D: ROBOTTITESTIN LOKITIEDOSTO KIRJAUTUMISMALLISTA



4/29/2019 Mbt Test Log

Mbt Test Log

Generated
20190423 15:59:50 UTC+03:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:10:22	
All Tests	1	1	0	00:10:22	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Mbt Test	1	1	0	00:10:23	

Test Execution Log

SUITE **Mbt Test** 00:10:22.901

Full Name: Mbt Test
Documentation: A test suite for MBT testing
This test has a workflow that is created using keywords in the imported resource file.
Source: C:\Users\S5916\Koodaus_local\Malli\pohjaisuus\Testit\mbt_test.robot
Start / End / Elapsed: 20190423 15:49:26.924 / 20190423 15:59:49.825 / 00:10:22.901
Status: 1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed

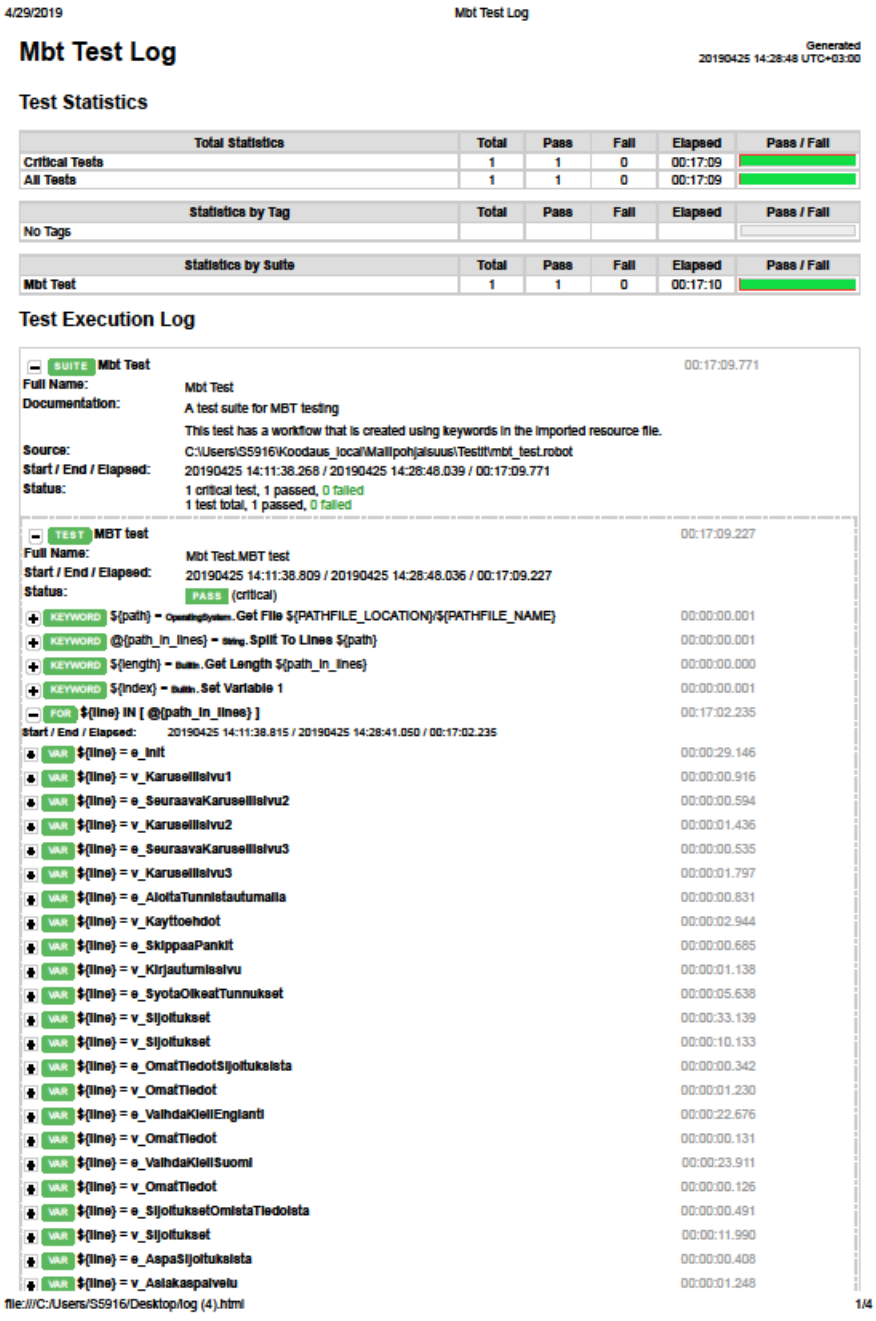
TEST **MBT test** 00:10:22.472

Full Name: Mbt Test.MBT test
Start / End / Elapsed: 20190423 15:49:27.351 / 20190423 15:59:49.823 / 00:10:22.472
Status: PASS (critical)

KEYWORD ${path} = OperatingSystem . **Get File** ${PATHFILE_LOCATION}/${PATHFILE_NAME} 00:00:00.001
KEYWORD @{path_in_lines} = String . **Split To Lines** ${path} 00:00:00.001
KEYWORD ${length} = Builtin . **Get Length** ${path_in_lines} 00:00:00.000
KEYWORD ${index} = Builtin . **Set Variable** 1 00:00:00.000
FOR **${line} IN [@{path_in_lines}]** 00:10:17.303
Start / End / Elapsed: 20190423 15:49:27.356 / 20190423 15:59:44.659 / 00:10:17.303
VAR ${line} = e_Init 00:00:16.104
VAR ${line} = v_Aloitus 00:00:00.663
VAR ${line} = e_PainaKirjaudu 00:00:00.562
VAR ${line} = v_Kirjautumissivu 00:00:01.810
VAR ${line} = e_SyotaVaaraTunnus 00:00:02.253
VAR ${line} = v_Kirjautumisvirhe 00:00:00.936
VAR ${line} = e_PalaaAlkuunKirjautumisvirheesta 00:00:00.270
VAR ${line} = v_Aloitus 00:00:01.334
VAR ${line} = e_PainaInfoKuvaketta 00:00:00.553
VAR ${line} = v_InfoSivu 00:00:00.951
VAR ${line} = e_PalaaInfoetaAlkuun 00:00:00.313
VAR ${line} = v_Aloitus 00:00:01.023
VAR ${line} = e_PainaOtaKayttoon 00:00:00.646
VAR ${line} = v_Kayttoehdot 00:00:11.172
VAR ${line} = e_HyvaksyKayttoehdot 00:00:06.825
VAR ${line} = v_PankkiTunnistautuminen 00:00:16.099
VAR ${line} = e_ValitseTupasPankki 00:00:06.414
VAR ${line} = v_Tupas 00:00:00.684
VAR ${line} = e_PainaValmisTupas 00:00:10.649
VAR ${line} = v_Aloitus 00:00:00.400
VAR ${line} = e_PainaOtaKayttoon 00:00:00.815
VAR ${line} = v_Kayttoehdot 00:00:09.407
VAR ${line} = e_HyvaksyKayttoehdot 00:00:06.972

file:///C:/Users/S5916/Desktop/log (3).html 1/4

LIITE E: ROBOTTITESTIN LOKITIEDOSTO KIRJAUTUMISESTA JA SOVELLUKSEN RUNGOSTA



4/29/2019 Mbt Test Log

Mbt Test Log

Generated
20190425 14:28:48 UTC+03:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:17:09	
All Tests	1	1	0	00:17:09	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Mbt Test	1	1	0	00:17:10	

Test Execution Log

SUITE **Mbt Test** 00:17:09.771

Full Name: Mbt Test
Documentation: A test suite for MBT testing
This test has a workflow that is created using keywords in the imported resource file.
Source: C:\Users\S5916\Koodaus_local\Mallipohjaisuus\Testit\mbt_test.robot
Start / End / Elapsed: 20190425 14:11:38.268 / 20190425 14:28:48.039 / 00:17:09.771
Status: 1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed

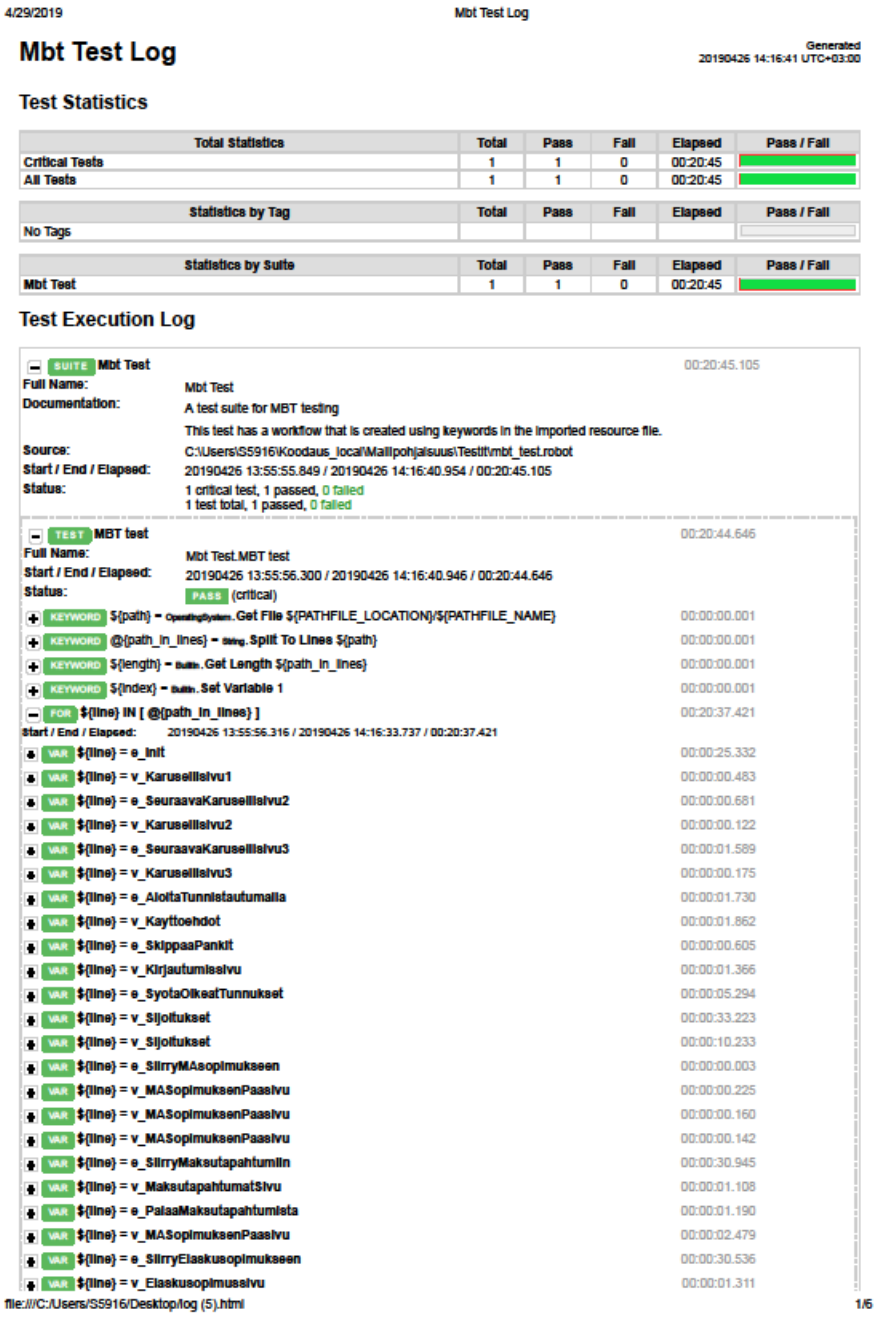
TEST **MBT test** 00:17:09.227

Full Name: Mbt Test.MBT test
Start / End / Elapsed: 20190425 14:11:38.809 / 20190425 14:28:48.036 / 00:17:09.227
Status: PASS (critical)

KEYWORD ${path} = OperatingSystem . **Get File** ${PATHFILE_LOCATION}/${PATHFILE_NAME} 00:00:00.001
KEYWORD @{path_in_lines} = String . **Split To Lines** ${path} 00:00:00.001
KEYWORD ${length} = Builtin . **Get Length** ${path_in_lines} 00:00:00.000
KEYWORD ${index} = Builtin . **Set Variable** 1 00:00:00.001
FOR **${line} IN [@{path_in_lines}]** 00:17:02.235
Start / End / Elapsed: 20190425 14:11:38.815 / 20190425 14:28:41.050 / 00:17:02.235
VAR ${line} = e_Init 00:00:29.146
VAR ${line} = v_KaruselliSivu1 00:00:00.916
VAR ${line} = e_SeuraavaKaruselliSivu2 00:00:00.594
VAR ${line} = v_KaruselliSivu2 00:00:01.436
VAR ${line} = e_SeuraavaKaruselliSivu3 00:00:00.535
VAR ${line} = v_KaruselliSivu3 00:00:01.797
VAR ${line} = e_AloitaTunnistautumalla 00:00:00.831
VAR ${line} = v_Kayttoehdot 00:00:02.944
VAR ${line} = e_SkippaaPankit 00:00:00.685
VAR ${line} = v_KirjautumisSivu 00:00:01.138
VAR ${line} = e_SyotaOikeatTunnukset 00:00:05.638
VAR ${line} = v_SijoitukseT 00:00:33.139
VAR ${line} = v_SijoitukseT 00:00:10.133
VAR ${line} = e_OmatTiedotSijoituksista 00:00:00.342
VAR ${line} = v_OmatTiedot 00:00:01.230
VAR ${line} = e_VaihdaKieliEnglanti 00:00:22.676
VAR ${line} = v_OmatTiedot 00:00:00.131
VAR ${line} = e_VaihdaKieliSuomi 00:00:23.911
VAR ${line} = v_OmatTiedot 00:00:00.126
VAR ${line} = e_SijoituksetOmistaTiedoista 00:00:00.491
VAR ${line} = v_SijoitukseT 00:00:11.990
VAR ${line} = e_AspaSijoituksista 00:00:00.408
VAR ${line} = v_Asiakaspalvelu 00:00:01.248

file:///C:/Users/S5916/Desktop/log (4).html 1/4

LIITE F: ROBOTTITESTIN LOKITIEDOSTO KIRJAUTUMISESTA JA VAURASTUMISEN PALVELUN SOPIMUKSESTA



4/29/2019 Mbt Test Log

Mbt Test Log

Generated
20190426 14:16:41 UTC+03:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:20:45	
All Tests	1	1	0	00:20:45	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Mbt Test	1	1	0	00:20:45	

Test Execution Log

SUITE **Mbt Test** 00:20:45.105

Full Name: Mbt Test
Documentation: A test suite for MBT testing
This test has a workflow that is created using keywords in the imported resource file.
Source: C:\Users\S5916\Koodaus_local\Mallipohjaisuus\Testit\mbt_test.robot
Start / End / Elapsed: 20190426 13:55:55.849 / 20190426 14:16:40.954 / 00:20:45.105
Status: 1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed

TEST **MBT test** 00:20:44.646

Full Name: Mbt Test.MBT test
Start / End / Elapsed: 20190426 13:55:56.300 / 20190426 14:16:40.946 / 00:20:44.646
Status: PASS (critical)

KEYWORD ${path} = OperatingSystem . **Get File** ${PATHFILE_LOCATION}/${PATHFILE_NAME} 00:00:00.001
KEYWORD @{path_in_lines} = String . **Split To Lines** ${path} 00:00:00.001
KEYWORD ${length} = Builtin . **Get Length** ${path_in_lines} 00:00:00.001
KEYWORD ${index} = Builtin . **Set Variable** 1 00:00:00.001
FOR **${line} IN [@{path_in_lines}]** 00:20:37.421
Start / End / Elapsed: 20190426 13:55:56.316 / 20190426 14:16:33.737 / 00:20:37.421
VAR ${line} = e_Init 00:00:25.332
VAR ${line} = v_Karusellisivu1 00:00:00.483
VAR ${line} = e_SeuraavaKarusellisivu2 00:00:00.681
VAR ${line} = v_Karuselliisivu2 00:00:00.122
VAR ${line} = e_SeuraavaKarusellisivu3 00:00:01.589
VAR ${line} = v_Karuselliisivu3 00:00:00.175
VAR ${line} = e_AloitaTunnistautumalla 00:00:01.730
VAR ${line} = v_Kayttoehdot 00:00:01.862
VAR ${line} = e_SkippaaPankit 00:00:00.605
VAR ${line} = v_Kirjautumissivu 00:00:01.366
VAR ${line} = e_SyotaOikeatTunnukset 00:00:05.294
VAR ${line} = v_Sijoitukset 00:00:33.223
VAR ${line} = v_Sijoitukset 00:00:10.233
VAR ${line} = e_SiirryMAsopimukseen 00:00:00.003
VAR ${line} = v_MASopimuksenPaasivu 00:00:00.225
VAR ${line} = v_MASopimuksenPaasivu 00:00:00.160
VAR ${line} = v_MASopimuksenPaasivu 00:00:00.142
VAR ${line} = e_SiirryMaksutapahtumiin 00:00:30.945
VAR ${line} = v_MaksutapahtumatSivu 00:00:01.108
VAR ${line} = e_PalaaMaksutapahtumista 00:00:01.190
VAR ${line} = v_MASopimuksenPaasivu 00:00:02.479
VAR ${line} = e_SiirryElaskusopimukseen 00:00:30.536
VAR ${line} = v_Elaskusopimussivu 00:00:01.311